PnP: Parallel And External Memory Iceberg Cube Computation

Ying Chen
Dalhousie University
Halifax, Canada
ychen@cs.dal.ca

Frank Dehne
Griffith University
Brisbane, Australia
www.dehne.net

Todd Eavis
Concordia University
Montreal, Canada
toddeavis@rogers.com

Andrew Rau-Chaplin
Dalhousie University
Halifax, Canada
www.cs.dal.ca/∼arc

Abstract

Motivated by the work of Ng et.al., and the recent success of Star-Cubing (Han et.al.), we further investigate the use of hybrid approaches for the parallel computation of very large iceberg-cube queries. We present "Pipe 'n Prune" (PnP), a new hybrid method for iceberg-cube query computation. The novelty of our method is that it achieves a tight integration of top-down piping for data aggregation with bottom-up Apriori data pruning. A particular strength of PnP is that it is very efficient for all *of the following scenarios: (1) Sequential iceberg-cube queries. (2) External memory iceberg-cube queries. (3) Parallel iceberg-cube queries on shared-nothing PC clusters with multiple disks.*

We performed an extensive performance analysis of PnP for all of the above scenarios with the following main results: In the first scenario, PnP performs very well for both, dense and *sparse data sets, providing an interesting alternative to BUC and Star-Cubing. In the second scenario, PnP shows a surprisingly efficient handling of disk I/O, with an external memory running time that is less than twice the running time for full in-memory computation of the same iceberg-cube query. In the third scenario, PnP scales very well, providing near linear speedup for a larger number of processors, thereby solving an open scalability problem observed by Ng et.al.*

1 Introduction

One of the most powerful and prominent technologies for knowledge discovery in Decision Support Systems (DSS) environments is On-line Analytical Processing (OLAP) [6]. By exploiting multi-dimensional views of the underlying *data warehouse*, the OLAP server allows users to "drill down" or "roll up" on hierarchies, "slice and dice" particular attributes, or perform various statistical operations such as ranking and forecasting. To support this functionality, OLAP relies heavily upon a data model known as the *data cube* [17, 18]. Conceptually, the data cube allows users to view organizational data from different perspectives and at a variety of summarization levels. It consists of the *base cuboid*, the finest granularity view containing the full complement of d dimensions (or attributes), surrounded by a collection of $2^d - 1$ sub-cubes/cuboids that represent the aggregation of the base cuboid along one or more dimensions. The data cube *operator* (an SQL syntactical extension) was proposed by Gray et al. [17] as a means of simplifying the process of data cube construction. Subsequent to the publication of the seminal data cube paper, a number of independent research projects began to focus on designing efficient algorithms for the computation of the data cube [3, 2, 18, 20, 21, 23, 24, 25, 26, 27, 28, 30, 31].

The size of data cubes can be massive. In the Winter Corporation's report [29], the largest three DSS databases exceed 20 Terabytes in size. More importantly, it is expected that by the end of 2004, the storage requirements of more than 40% of production data warehouses will exceed one Terabyte [12]. One approach for dealing with the data cube size is to allow user-specific constraints. For iceberg-cubes (e.g. [3, 13, 30]), aggregate values are only stored if they have a certain, user specified, minimum support. Another possible approach is to introduce parallel processing which can provide two key ingredients for dealing with the data cube size: increased computational power through multiple processors and increased I/O bandwidth through multiple parallel disks (e.g. [4, 5, 7, 8, 9, 10, 14, 15, 16, 19, 22]). In [23], Ng et.al. combined both of the above approaches and studied various algorithms for parallel iceberg-cube computation on PC clusters. The algorithm of choice in [23], referred to as PT, applies a *hybrid* approach in that it combines top-down data aggregation with bottom-up data reduction.

Motivated by the work of Ng et.al. [23] and the recent success of another hybrid sequential method, Star-Cubing [30], we further investigate the use of hybrid approaches for the *parallel* computation of iceberg-cube queries. We present a new hybrid method, called "Pipe 'n Prune" (PnP), for iceberg-cube query computation. Our approach combines top-down data aggregation through piping with bottom-up Apriori data reduction. The main difference to previous approaches is the introduction of a novel PnP operator which uses a piping approach to aggregate data and, at the same time, performs Apriori pruning for subsequent group-by computations. Our approach was motivated by the work of Ng et.al. [23] who presented a two phase hybrid parallel method, PT, which first partitions BUC bottom-up computation and then use top-down aggregation for building the startup group-by for each partition. Inspired by Star-Cubing [30], our new PnP operator extends this two phase approach towards a complete merge between data aggregation and Apriori pruning. PnP is very different from Star-Cubing [30] in that PnP retains top-down data aggregation through piping and interleaves it with iceberg bottom-up data reduction (pruning). An illustration of our approach is sketched in Figure 2. An important property of our PnP method is that it is composed mainly of linear data scans and does not require complex in-memory structures. This allows us to extend PnP to external memory computation of very large iceberg-cube queries with only minimal loss of efficiency. In addition, PnP is well suited for shared-nothing parallelization (where processors do not share any memory and all data is partitioned and distributed over a set of disks). Our new parallel, external memory, PnP method provides close to linear speedup particularly on those data sets that are hard to handle for sequential methods. In addition, parallel PnP scales well and provides linear speedup for larger number of processors, thereby also solving an open scalability problem observed in [23].

This paper makes the following contributions:

- We present a novel PnP operator and "Pipe 'n Prune" (PnP) algorithm for the computation of iceberg-cube queries. The novelty of our method is that it completely interleaves a top-down piping approach for data aggregation with bottom-up Apriori data pruning. A particular strength of PnP is that it is very efficient for *all* of the following scenarios:
  - Sequential iceberg-cube queries.
  - External memory iceberg-cube queries.
  - Parallel iceberg-cube queries on shared-nothing PC clusters with multiple disks.
- We performed an extensive performance analysis of PnP for all of the above scenarios. In general, PnP performs very well for both, dense *and* sparse data sets and it scales well, providing linear speedup for larger number of processors. In [23] Ng et.al. observe for their parallel iceberg-cube method that "the speedup from 8 processors to 16 processors is below expectation" and attribute this scalability problem to scheduling and load balancing issues. Our analysis shows that PnP solves these problems and scales well for at least up to 16 processors.

  In more detail, our analysis of PnP for the above three scenarios showed the following:

  - **Sequential iceberg-cube queries:** As a special case, PnP also provides a new *sequential* hybrid method for the computation of iceberg-cube queries. We present an extensive performance analysis of PnP in comparison with BUC [3] and StarCube [30]. We observe that the sequential performance of PnP is very stable even for large variations of data density and data skew. Sequential PnP typically shows a performance between BUC and StarCube, while BUC and StarCube have ranges of data density and skew where BUC outperforms StarCube or vice versa. For the special case of full cube computation, PnP outperforms both BUC and StarCube.
  - **External memory iceberg-cube queries:** Since PnP is composed mainly of linear scans and does not require complex in-memory data structures, it is conceptually easy to implement as an external memory method for very large iceberg-cube queries. In order to make good use of PnP's properties, we have implemented our own I/O manager to have full control over latency hiding through overlapping of computation and disk I/O. We present an extensive performance analysis of PnP for external memory computation of very large iceberg-cube queries. Our experiments show minimum loss of efficiency when PnP switches from in-memory to external memory computation. The measured external memory running time (where PnP is forced to use external memory by limiting the available main memory) is less than twice the running time for full in-memory computation of the same iceberg-cube query.
  - **Parallel iceberg-cube queries on shared-nothing PC clusters (with multiple disks):** PnP is well suited for shared-nothing parallelization, where processors do not share any memory and all data is partitioned and distributed over a set of disks (see Figure 4). We present a PnP parallelization which (1) minimizes communication overhead, (2) balances work load, and (3) makes full use of our I/O manager by overlapping *parallel* computation and *parallel* disk access on all available disks in the PC cluster. Extensive experiments show that our new parallel, external memory, PnP method provides close to linear speedup particularly on those data sets that are hard to handle for sequential methods. Most importantly, parallel PnP scales well and provides

near linear speedup for larger numbers of processors, thereby also solving an important open scalability problem observed in [23].

The remainder of this paper is organized as follows. Section 2 provides first a high level overview of our PnP approach and then present the algorithmic details for the three scenarios mentioned above. Section 3 presents an in-depth performance evaluation of PnP, and Section 4 concludes our paper.

2 The PnP Algorithm

PnP is a hybrid, sort-based, algorithm for the computation of very large iceberg-cube queries. The idea behind PnP is to fully integrate data aggregation via top-down piping [26] with bottom-up (BUC [3]) Apriori pruning. We introduce a new operator, called the *PnP operator*. For a group-by v, the PnP operator performs two steps: (1) It builds all group-bys v' that are a prefix of v through one single sort/scan operation (piping [26]) with iceberg-cube pruning. (2) It uses these prefix group-bys to perform bottom-up (BUC [3]) Apriori pruning for new group-bys that are starting points of other piping operations. An example of a 5-dimensional PnP operator is shown in Figure 1. The PnP operator is applied recursively until all group-bys of the iceberg-cube have been generated. An example of a 5-dimensional *PnP Tree* depicting the entire process for a 5-dimensional iceberg-cube query is shown in Figure 2. The remainder of this section describes in detail our PnP method for the following three scenarios:

- Sequential, in memory, iceberg-cube queries.
- External memory iceberg-cube queries.
- Parallel iceberg-cube queries on shared-nothing PC clusters with multiple disks.

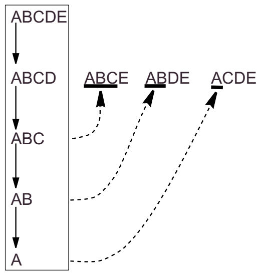

Figure 1. A PnP Operator.

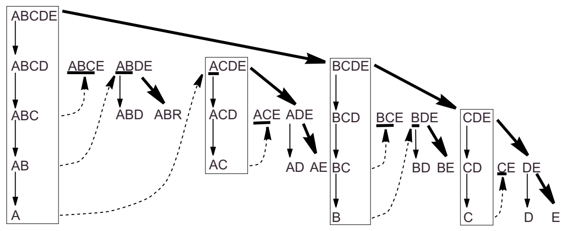

Figure 2. A PnP Tree. (Plain arrow: Top-Down Piping. Dashed Arrow: Bottom-up Pruning. Bold Arrow: Sorting.)

2.1 PnP: Sequential In-Memory Version

We assume as input a table $R[1..n]$ representing a d-dimensional raw data set R consisting of n rows $R[i]$, $i = 1 \ldots n$. Because of the iceberg-cube constraint, a cell in a cuboid is only returned if it has *minimum support*. That is, a cell is only calculated if there are at least min_sup tuples assigned to that cell, for some given input parameter min_sup.

For a row $R[i]$ we denote with $\underline{R}_j[i]$ the prefix of $R[i]$ consisting of the first j feature values of $R[i]$, followed by the measure value of $R[i]$. We denote with $\hat{R}^j[i]$ the row $R[i]$ with its feature value in dimension j removed. We denote with $\emptyset$ the empty (0-dimensional) group-by. For a group-by v we denote with $|v|$ the number of dimensions of v, and with $\hat{v}^j$ the group-by that is the same as v but with dimension j removed. We denote with $\underline{v}_j$ the group-by identifier consisting of the first j dimensions of v.

Our PnP method for the sequential, in memory, case is shown in Algorithms 1 and 2. Algorithms 2 represents the main part, the implementation of the recursive PnP operator.

We explain our algorithm using the example in Figure 2 for a 5-dimensional iceberg-cube query. In Line 2 of Algorithms 1, we call PnP-1(R, ABCDE, $\emptyset$). This will first result in the creation of the pipe ABCDE - ABCD - ABC - AB - A and then create pruned versions of ABCE, ABDE, and ACDE for subsequent piping operations. Table 1 shows a complete execution for the example raw data set R indicated in the first column of Table 1. Buffers $b[5]$ ... $b[1]$ represent the results of piping operations, while R_3 ... R_1 show the result of pruning operations. Note that, the PnP operator uses only one single pass through the data set. The horizontal lines in Table 1 indicate cases where aggregation or pruning take place. The recursive call in Line 12 of Algorithms 2 initiates the PnP operator for group-bys ABCE, ABDE, and ACDE. The prefix passed as third parameter in Line 12 of Algorithms 2 is shown in Figure 2 as the underlined portions of ABCE, ABDE, and ACDE, respectively. It represents for those recursive calls the portion of the pipe

R	$b[5]$	$b[4]$	$b[3]$	R_3	$b[2]$	R_2	$b[1]$	R_1
ABCDE	**ABCDE**	**ABCD**	**ABC**	**ABCE**	**AB**	**ABDE**	**A**	**ACDE**
11111 1	11111 1			pruned		1111 2		1111 1
11112 1	11112 1	1111 2				1112 1		1112 1
11122 1	11122 1	1112 1	111 3			1122 1		1122 1
11211 1	11211 1	1121 1	112 1	pruned	**11 4**		**1 4**	1211 1
21111 1	21111 1	2111 1		pruned		pruned		pruned
21121 1	21121 1							
21122 1	21122 1	2112 2	211 3		21 3		2 3	
31111 1	31111 1	3111 1		pruned		3111 1		3111 1
31121 1	31121 1					3121 2		3121 1
31122 1	31122 1	3112 2	311 3			3122 1		3122 1
31221 1	31221 1			pruned		3123 1		3221 1
31223 1	31223 1	3122 2	312 2		**31 5**		**3 5**	3223 1
41111 1	41111 1	4111 1	411 1	pruned	41 1	pruned		4111 2
42111 1	42111 1			pruned		pruned		4112 1
42112 1	42112 1	4211 2	421 2		42 2			4121 1
43121 1	43121 1	4312 1	431 1	pruned	43 1	pruned	**4 4**	

Table 1. PnP Processing of $ABCDE$

that has already been computed. The recursive call in Line 20 of Algorithms 2 initiates the PnP operator for group-by BCDE and starts the iceberg-cube computation for all group-bys not containing A. The resulting entire process is depicted in Figure 2.

Algorithm 1 Algorithm PnP: sequential, in memory

```
Input: $R[1..n]$: a table representing a $d$-dimensional raw data set consisting of $n$ rows $R[i]$, $i = 1 \dots n$; $min\_sup$: the minimum support.
Output: The iceberg data cube.
1: Sort $R$ and aggregate duplicates in $R$.
2: Call PnP-1($R$, $v_R$, $\emptyset$), where $v_R$ is the group-by containing all dimensions of $R$ (sorted by cardinality in decreasing order).
```

2.2 PnP: Sequential External Memory Version

Since PnP is sort based, it is easy to extend PnP to external memory, as shown in Algorithms 3 and 4. We discuss here only the main differences between Algorithm 2 and Algorithm 4. All sort operations are replaced by external memory sorts. Some care has to be taken with the scan and aggregation/pruning operations, as buffers may overflow and have to be saved to disk. The main difference between Algorithm 2 and Algorithm 4 is with respect to the recursive calls in Line 12 in Algorithm 2. In the external memory version, we have to save the tables R_j into a file F_j on disk as shown in Line 11 of Algorithm 4. A separate loop in Lines 18 to 22 of Algorithm 4 is then required to retrieve all R_j and perform the recursive calls. Note that, these operations are independent and we can apply disk latency hiding through overlapping of computation and disk I/O. In order to make good use of this effect, we have implemented our own I/O manager which resulted in a significant performance improvement (see Section 3.2).

Algorithm 2 PnP-1(R, v, pv)

```
Input: $R[1..n]$: a table representing the raw data set consisting of $n$ rows $R[i]$, $i = 1 \dots n$; $v$: identifier for a group-by of $R$; $pv$: a prefix of $v$.
Output: The iceberg data cube.
1: Local Variables: $k = |v| - |pv|$; $R_j$: tables for storing rows of $R$; $b[1..k]$: a buffer for storing $k$ rows, one for each group-by $\underline{v}_1 \dots \underline{v}_k$; $h[1..k]$: $k$ integers; $i$, $j$: integer counters. Initialization: $b[1..k]$ = [null .. null]; $h[1..k]$ = $[1..1]$.
2: for $i = 1..n$ do
3:   for $j = k..1$ do
4:     if ($b[j]$ = null) OR (the feature values of $b[j]$ are a prefix of $R[i]$) then
5:       Aggregate $\underline{R}_j[i]$ into $b[j]$.
6:     else
7:       if $b[j]$ has minimum support then
8:         Output $b[j]$ into group-by $\underline{v}_j$.
9:         if $j \leq k - 2$ then
10:          Create a table $R_j = \hat{R}^{j+1}[h[j]] \dots \hat{R}^{j+1}[i-1]$.
11:          Sort and aggregate $R_j$.
12:          Call PnP-1($R_j$, $\hat{v}^{j+1}$, $\underline{v}_j$).
13:        end if
14:      end if
15:      Set $b[j] = null$ and $h[j] = i$.
16:    end if
17:  end for
18: end for
19: Create a table $R'[1..n']$ by sorting and aggregating $\hat{R}^1[1] \dots \hat{R}^1[n]$.
20: Call PnP-1($R'$, $\hat{v}^1$, $\emptyset$).
```

Algorithm 3 Algorithm PnP: sequential, external memory

```
Input: $R[1..n]$: a table (stored on disk) representing a $d$-dimensional raw data set consisting of $n$ rows $R[i]$, $i = 1 \dots n$; $min\_sup$: the minimum support.
Output: The iceberg data cube (stored on disk).
1: Sort $R$, using external memory sorting, and aggregate duplicates in $R$.
2: Call PnP-2($R$, $v_R$, $\emptyset$), where $v_R$ is the group-by containing all dimensions $R$ (sorted by cardinality in decreasing order).
```

Algorithm 4 PnP-2(R, v, pv)

Input: $R[1..n]$: a table (stored on disk) representing the raw data set consisting of n rows $R[i]$, $i = 1 \ldots n$; v: identifier for a group-by of R; pv: a prefix of v.

Output: The iceberg data cube (stored on disk).

```
1:  Local Variables: $k = |v| - |pv|$; $R_j$: tables for storing rows of $R$ (called partitions); $F_j$: disk files for storing multiple partitions $R_j$; $b[1..k]$: a buffer for storing $k$ rows, one for each group-by $\underline{v}_1 \ldots \underline{v}_k$; $h[1..k]$: $k$ integers; $i$, $j$: integer counters. Initialization: $b[1..k]$ = [null .. null]; $h[1..k]$ = [1..1].
2:  for $i = 1..n$ (while reading $R[i]$ from disk in streaming mode...) do
3:    for $j = k..1$ do
4:      if ($b[j]$ = null) OR (the feature values of $b[j]$ are a prefix of $R[i]$) then
5:        Aggregate $\underline{R}_j[i]$ into $b[j]$.
6:      else
7:        if $b[j]$ has minimum support then
8:          Output $b[j]$ into group-by $\underline{v}_j$. Flush to disk if $\underline{v}_j$'s buffer is full.
9:          if $j \leq k - 2$ then
10:           Create a table $R_j$ = $\hat{R}^{j+1}[h[j]]$ ... $\hat{R}^{j+1}[i-1]$.
11:           Sort and aggregate $R_j$ (using external memory sort if necessary). Write the resulting $R_j$ and an "end-of-partition" symbol to file $F_j$.
12:         end if
13:       end if
14:       Set $b[j] = null$ and $h[j] = i$.
15:     end if
16:   end for
17: end for
18: for $j = k..1$ do
19:   for each partition $R_j$ written to disk file $F_j$ in line 11 do
20:     Call PnP-2($R_j$, $\hat{v}^{j+1}$, $\underline{v}_j$).
21:   end for
22: end for
23: Create a table $R'[1..n']$ by sorting and aggregating $\hat{R}^1[1]$ ... $\hat{R}^1[n]$ (using external memory sort if necessary).
24: Call PnP-2($R'$, $\hat{v}^1$, $\emptyset$).
```

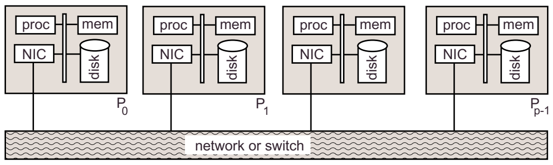

Figure 3. Shared-Nothing Multiprocessor

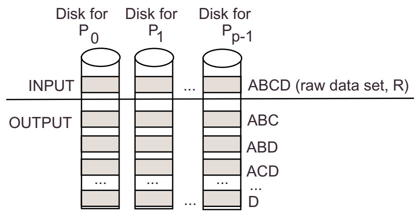

Figure 4. Parallel Disk Layout.

2.3 PnP: Parallel And External Memory Version

We now discuss how our PnP algorithm can be parallelized in order to be executed on a shared-nothing multiprocessor as shown in Figure 3. Such a multiprocessor consists of p processors P_0 ... P_{p-1}, each with its own memory and disk. The processors are connected via a network or switch. Our focus is on practical parallel methods that can be implemented on low-cost, *Beowulf* style, PC clusters consisting of standard Intel processor based Linux machines connected via Gigabit Ethernet. However, our methods can also be used, and will perform even better, on more expensive platforms such as clusters connected via Myrinet or shared memory parallel machines like the SunFire.

We assume as input a d-dimensional raw data set R stored in a table consisting of n rows that are distributed over the p processors as shown in Figure 4. More precisely, every processor P_i stores on its disk a table R_i consisting of $\frac{n}{p}$ rows of R. As indicated in Figure 4, each group-by of the output (iceberg-cube) will also be partitioned and distributed over the p processors. We refer to this process as *striping* a table over the p disks. When every group-by is striped over the p disks, access to the group-by can be performed with maximum I/O bandwidth through full parallel disk access.

For a d-dimensional table R_i, we define tables T_i^j, $j = 1, .., d$, as the tables obtained by removing from each row of R_i the first $j - 1$ feature values and performing aggregation to remove duplicates (but not performing iceberg-cube pruning). Note that, $T_i^1 = R_i$.

Our parallel PnP method is shown in Algorithm 5. The basic idea is illustrated in Figure 5. The figure shows a *PnP forest* obtained by partitioning the PnP tree in Figure 2 into d trees, one for each feature dimension. The data set for the root of the j-th tree is the set $T^j = T_0^j \cup T_1^j \cup ... \cup T_{p-1}^j$. We start with $T^1 = R$ striped over the p disks, where processor P_i stores $T_i^1 = R_i$, and execute on each processor P_i the sequential Algorithm 3 with input T_i^1 (Line 7 of Algorithm 5). This creates the first tree in the PnP forest of Figure 5. Next, we compute on each processor P_i the table T_i^2 from T_i^1 by removing the first feature dimension and performing aggregation to remove duplicates (via a sequential sort); see Line 3 of Algorithm 5. Different data aggregation on different processors can lead to imbalance between processors, and the set $T_0^2 \cup T_1^2 \cup ... \cup T_{p-1}^2$ is therefore rebalanced through a global sort (Line 4 of Algorithm 5). We can then execute on each processor P_i the sequential Algorithm 3 with input T_i^2 (Line 7 of Algorithm 5), creating the second tree in the PnP forest of Figure 5. This process is iterated d times, until all group-bys have been built.

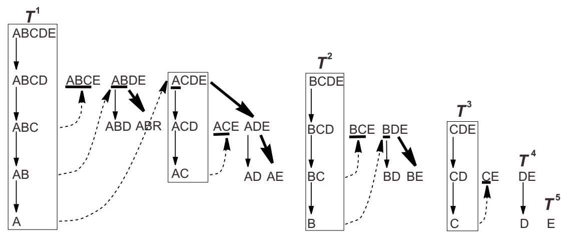

Figure 5. A PnP Forest.

Algorithm 5 Algorithm PnP: parallel, external memory

```
Input: $R$: a table representing a $d$-dimensional raw data
    set consisting of $n$ rows, stored on $p$ processors. Every
    processor $P_i$ stores (on disk) a table $R_i$ of $\frac{n}{p}$ rows of $R$
    as shown in Figure 4. $min\_sup$: the minimum support.
Output: The iceberg data cube (distributed over the disks
    of the $p$ processors as shown in Figure 4).
1: Variables: On each processor $P_i$ a set of $d$ tables $T_i^1$,
   ..., $T_i^d$.
2: for $j = 1..d$ do
3:     Each processor $P_i$: Compute $T_i^j$ from $T_i^{j-1}$ via sequential sort. ($T_i^1 = R_i$)
4:     Perform a parallel global sort on $T_1^j \cup T_1^j \cup ... \cup T_p^j$.
5: end for
6: for $j = 1..d$ do
7:     Each processor $P_i$: Apply Algorithm 3 to $T_i^j$.
8: end for
```

3 Performance Evaluation

We have implemented the sequential, external memory, and parallel versions of our PnP algorithm as presented in the previous section. Our sequential C++ code evolved from the code for top-down sequential pipesort used in [4]. Our external memory code evolved from the sequential code through the addition of an external memory sort and a specially written I/O manager that allows us to overlap computation and disk I/O. Our parallel code evolved, in turn, from our the external memory code through the addition of communication operations drawn from the MPI communication library.

Our performance evaluation was conducted in two stages. In the first stage we evaluate the sequential version of PnP by comparing it with implementations of BUC and Star-Cubing. The Microsoft Window's executables for these implementations were kindly provided by J. Han's research group to enable just such comparative performance testing of cube construction methods [1]. The PnP codes, for both the sequential (in-memory) version and external memory version, were compiled using Visual C++ 6.0. Both sequential and external memory experiments were conducted on a 2.8 GHz Intel Pentium 4 based PC running Microsoft Windows 2000 with 1 GB RAM and an 80 GB 7200 RPM IDE disk.

In the second stage of our performance evaluation we explored the performance of our the parallel version of PnP on a 32 processor Beowulf style cluster. This shared nothing parallel machine consists of a collection of 1.7 GHz Intel Xeon processors each with 1 GB of RAM, two 40 GB 7200 RPM IDE disks and an onboard Inter Pro 1000 XT NIC. Each processor is running Linux Redhat 7.2 with gcc 2.95.3 and MPI/LAM 6.5.6. as part of a ROCKS cluster distribution. All processors are interconnected via a Cisco 6509 GigE switch. Due to restrictions in machine access, we were frequently unable to reserve all 32 processors of this machine. In such cases a minimum of 16 processors were used.

In the following experiments, all sequential times are measured as wall clock times in seconds. All parallel times are measured as the wall clock time between the start of the first process and the termination of the last process. All times include the time taken to read the input from files and write the output into files. Furthermore, all wall clock times are measured with no other users on the machine. The running times for BUC and Star-Cubing that we show are those captured and reported by the executables obtained from [1].

To fully explore the performance of these cube construction methods we generated a large number of synthetic data sets which varied in terms of the following parameters: n - the number of rows in the raw data set R, d - the number of dimensions, s - the skew in each dimension as a zifp value, m - the minimum support, b - the available memory in bytes, and $c_1 \dots c_d$ - the cardinality of each dimension (where an unsubscripted c indicates the same cardinality in all dimensions). The data generator used in the sequential and external memory experiments to generate the raw data set R

was provided with the BUC and Star-Cubing executables from [1]. For the parallel experiments we generated similar synthetic data sets using our own data generator, which had been previously used in [4, 5].

3.1 Sequential Experiments

The performance results for our sequential experiments are shown in Figures 6 to 17. Figures 6 to 9 compare PnP to BUC and Star-Cubing, first for the special case of full cube computation, and then for iceberg cube computation on raw data sets of varying sparsity. The remaining sequential experiments (Figures 10 to 17) explore various settings of the parameters n, d, m and s for both sparse and dense cubes.

Figures 6 and 7 show for full cube computation (i.e. $m = 1$) results for PnP compared to BUC and Star-Cubing on various cardinalities and growing data sizes. Note that varying cardinality, while holding the other parameters constant, amounts to varying the sparsity. We observe that for the special case of full cube computation the sequential version of PnP performs better than BUC or Star-Cubing regardless of sparsity. In this case, PnP takes full advantage of pipeline processing and saves significant time by sharing sorts, while bottom-up Apriori data pruning is ineffectual.

Figures 8 and 9 compare PnP to BUC and Star-Cubing for iceberg cube computation while varying sparsity. We observe that the sequential performance of PnP is very stable even for large variations of data density. Sequential PnP typically shows a performance between BUC and Star-Cubing, while BUC and Star-Cubing have ranges of data density where BUC outperforms Star-Cubing or vice versa.

Finally, Figures 10 to 17 compare PnP to BUC and Star-Cubing for iceberg cube computation while varying input size t, dimensionality d, minimum support m, and skew s, for both dense ($c = 10$) and sparse ($c = 100$) cubes. Again, we observe that the sequential performance of PnP is highly stable. Sequential PnP performance is almost always between that of BUC and Star-Cubing, while BUC tends to perform best on relatively sparse data sets and Star-Cubing best on somewhat denser data sets. In many of these experiments, the shape of the curves for the various methods are quite similar making the constants, and therefore issues of implementation, critical. Overall, sequential PnP appears to be an interesting alternative to BUC and Star-Cubing especially in applications where performance stability over a wide range of input parameters is important.

3.2 External Memory Experiments

The performance results for the external memory version of PnP are shown in Figures 18 and 19. Note that since PnP is composed mainly of linear scans and does not require complex in-memory data structures, it is reasonably easy to implement as an external memory method for very large iceberg-cube queries. For these experiments, in order to make good use of PnP's properties, we have implemented our own I/O manager to have full control over latency hiding through overlapping of computation and disk I/O.

In evaluating the external memory version of PnP we use larger data sets, ranging in size from 1 million to 20 million rows, while varying dimensionality d and available memory b.

Overall, our experiments show minimum loss of efficiency when PnP switches from in-memory to external memory computation. The measured external memory running time (where PnP is forced to use external memory by limiting the available main memory) is less than twice the running time for full in-memory computation of the same iceberg-cube query. In Figure 18 we observe similarly shaped curves even as we increase the dimensionality of the problem due in large part to the effects of iceberg pruning. The location of the slight jump in time, corresponding to the switch to external memory, occurs between 5 million rows and 7 million rows depending on the dimensionality of the iceberg cube being generated. Figure 19 shows, not surprisingly, that there is a benefit to increasing the memory space b available to the external memory algorithm. However, the relative size of this benefit diminishes significantly as t grows.

To test the scalability of our external memory version of PnP we also ran it on a number of extremely large input data sets. Without modification, PnP was able to construct iceberg cubes on input tables consisting of 200 Million rows (5.6 Gigabytes) and 6 dimensions in under three hours and forty-five minutes. To the best of our knowledge, PnP is the first cubing method described in the literature that has demonstrated the ability to construct iceberg cubes on such massive data sets.

3.3 Parallel Experiments

The performance results for the parallel shared-nothing version of PnP are shown in Figures 20 to 29. Note that this version is based on the external memory PnP code base and uses external memory processing, in addition to parallelism, as needed.

These experiments focus on *speedup*, that is they consist of incrementally increasing the number of processors available to the parallel version of PnP to determine the time and corresponding parallel speedup obtained while varying the other key parameters of input data size t, dimensionality d, cardinality c, minimum support m, and skew s. Speedup is one of the key metrics for the evaluation of parallel database systems [11] as it indicates the degree to which adding processors decreases the running time. The relative speedup for p processors is defined as $S_p = \frac{t_1}{t_p}$, where t_1 is the running time of the parallel program using one processor, all communication overhead having been removed from the program, and t_p is the running time using p processors. An ideal S_p is equal to p, which implies that p processors are p times faster than one processor. That is, the curve of an ideal S_p is a linear diagonal line.

Figure 20 shows the running time of parallel PnP for input data sizes between 1 and 8 million rows and Figure 21 shows the corresponding speedup. As is typically the case, relative speedup improves as we increase the size of the input and consequentially the total amount of work to be performed. With $t \geq 4,000,000$ rows, near optimal linear speedup is observed all the way up to 16 processors, while with $t = 2,000,000$ rows speedup drops off beyond 8 processors. In general, near linear speedup is observed when there is at least $n/p = 500,000$ rows per processor.

Figure 22 shows the running time of the parallel version of PnP for increasing dimensionality and Figure 23 shows the corresponding speedup. For this experiment we were able to reserves all 32 processors of our parallel machine. Again we observe near optimal linear speedup all the way up to 16 processor. With 32 processors the parallel version of PnP achieves at least 50% speedup when generating cubes of between 8 and 10 dimensions and near optimal linear speedup when generating a 11 dimensional cube. Note that the best speedup is achieved on the problems which are hardest to solve sequentially, that is those that involve the largest problems in terms of input size and/or dimensionality.

The cardinality of the dimensions in the input data can significantly effect performance. As cardinalities increase so does the sparsity of the data set, and this typically reduces the size of the resulting iceberg query result. Figure 24 shows the running time of the parallel version of PnP for input data covering a range of cardinalities and Figure 25 shows the corresponding speedup. Overall we observe near linear speedup for all but the most sparse data. When the cardinality is 512 in each of the 10 dimensions the data space is so sparse and the resulting iceberg cube so small that the speedup drops significantly. When the cardinality is 32 in each of the 10 dimensions the data space is so dense and the resulting iceberg cube so large that on a single processor significant external memory processing is required. In this case moving to more processors has the added advantage of avoiding much of the external memory processing and therefore in this case we actually observe super linear relative speedup.

Figures 26 to 29 show the effects on running time and speedup of varying minimum support and skew. We observe that for smaller values of minimum support and skew the time required is larger (as is typically the case in cube construction) and the speedup obtained by our parallel PnP algorithm is near linear. When either minimum support or skew are sufficiently large speedup falls off, however so does the wall clock time required by parallel PnP to compute the iceberg cube.

Overall our experiments show that the parallel, external memory, version of PnP provides close to linear speedup particularly on those data sets that are hard to handle for sequential methods. Most importantly, parallel PnP scales well providing near linear speedup for larger numbers of processors, thereby also solving an important open scalability problem observed in [23].

4 Conclusions

In this paper, we further investigated the use of hybrid approaches for the *parallel* computation of iceberg-cube queries and presented a new hybrid method, “Pipe ’n Prune” (PnP), for iceberg-cube query computation. The most important feature of our approach is that it completely interleaves top-down data aggregation through piping with bottom-up Apriori data reduction.

For sequential iceberg-cube queries, PnP typically shows a performance between BUC and StarCube, while BUC and StarCube have ranges of data density and skew where BUC outperforms StarCube or vice versa. This makes PnP an interesting new alternative method, especially in applications where performance stability over a wide range of input parameters is important. For external memory iceberg-cube queries, we observe minimum loss of efficiency. The measured external memory running time is less than twice the running time for full in-memory computation of the same iceberg-cube query. For parallel iceberg-cube queries on shared-nothing PC clusters, PnP scales well and provides near linear speedup for larger numbers of processors, thereby also solving an important open scalability problem observed in [23].

References

[1] J. Han, Software download site. http://www-sal.cs.uiuc.edu/∼hanj/pubs/software.htm.

[2] S. Agarwal, R. Agrawal, P. Deshpande, A. Gupta, J. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. *Proceedings of the 22nd International VLDB Conference*, pages 506–521, 1996.

[3] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. *Proceedings of the 1999 ACM SIGMOD Conference*, pages 359–370, 1999.

[4] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel rolap data cube construction on shared-nothing multiprocessors. *Distributed and Parallel Databases*, 15:219–236, 2004.

[5] Y. Chen, F.Dehne, T.Eavis, and A.Rau-Chaplin. Building large rolap data cubes in parallel. In *to appear in Proceedings of the 8th International Database Engineering and Applications Symposium (IDEAS '04)*, 2004.

[6] E. F. Codd. Providing olap (on-line analytical processing) to user-analysts: An it mandate. Technical report, E.F. Codd and Associates, 1993.

[7] F. Dehne, T. Eavis, S. Hambrusch, and A. Rau-Chaplin. Parallelizing the datacube. *International Conference on Database Theory*, 2001.

[8] F. Dehne, T. Eavis, and A. Rau-Chaplin. A cluster architecture for parallel data warehousing. *International Conference on Cluster Computing and the Grid (CCGRID 2001)*, 2001.

[9] F. Dehne, T. Eavis, and A. Rau-Chaplin. Computing partial data cubes for parallel data warehousing applications. *Euro PVM/MPI 2001*, 2001.

[10] F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallelizing the datacube. *Distributed and Parallel Databases*, 11(2):181–201, 2002.

[11] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[12] The Rising Storage Tide, 2003. http://www.datawarehousing.com/papers.

[13] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. Ullman. Computing iceberg queries efficiently. *in Proceedings VLDB*, pages 299–310, 1998.

[14] S. Goil and A. Choudhary. High performance OLAP and data mining on parallel computers. *Journal of Data Mining and Knowledge Discovery*, (4), 1997.

[15] S. Goil and A. Choudhary. High performance multidimensional analysis of large datasets. *Proceedings of the First ACM International Workshop on Data Warehousing and OLAP*, pages 34–39, 1998.

[16] S. Goil and A. Choudhary. A parallel scalable infrastructure for OLAP and data mining. *International Database Engineering and Application Symposium*, pages 178–186, 1999.

[17] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Proceeding of the 12th International Conference On Data Engineering*, pages 152–159, 1996.

[18] V. Harinarayan, A. Rajaraman, and J. Ullman. Implementing data cubes. *Proceedings of the 1996 ACM SIGMOD Conference*, pages 205–216, 1996.

[19] H.Lu, J. Yu, L. Feng, and X. Li. Fully dynamic partitioning: Handling data skew in parallel data cube computation. *Distributed and Parallel Databases*, 13:181–2002, 2003.

[20] L. Lakshmanan, J. Pei, and J. Han. Quotient cube: How to summarize the semantics of a data cube. *Proceedings of the 28th VLDB Conference*, 2002.

[21] L. Lakshmanan, J. Pei, and Y. Zhao. Qc-trees: An efficient summary structure for semantic olap. *Proceedings of the 2003 ACM SIGMOD Conference*, pages 64–75, 2003.

[22] S. Muto and M. Kitsuregawa. A dynamic load balancing strategy for parallel datacube computation. *ACM 2nd Annual Workshop on Data Warehousing and OLAP*, pages 67–72, 1999.

[23] R. Ng, A. Wagner, and Y. Yin. Iceberg-cube computation with PC clusters. *Proceedings of 2001 ACM SIGMOD Conference on Management of Data*, pages 25–36, 2001.

[24] K. Ross and D. Srivastava. Fast computation of sparse data cubes. *Proceedings of the 23rd VLDB Conference*, pages 116–125, 1997.

[25] N. Roussopoulos, Y. Kotidis, and M. Roussopolis. Cube-tree: Organization of the bulk incremental updates on the data cube. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 89–99, 1997.

[26] S. Sarawagi, R. Agrawal, and A.Gupta. On computing the data cube. Technical Report RJ10026, IBM Almaden Research Center, San Jose, California, 1996.

[27] Y. Sismanis, A. Deligiannakis, N. Roussopolos, and Y. Kotidis. Dwarf: Shrinking the petacube. *Proceedings of the 2002 ACM SIGMOD Conference*, pages 464–475, 2002.

[28] W. Wang, J. Feng, H. Lu, and J. Yu. Condensed cube: An effective approach to reducing data cube size. *Proceedings of the International Conference on Data Engineering*, 2002.

[29] The Winter Report. http://www.wintercorp.com/vldb/.

[30] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. *in Proceedings Int. Conf. on Very Large Data Bases (VLDB'03)*, 2003.

[31] Y. Zhao, P. Deshpande, and J. Naughton. An array-based algorithm for simultaneous multi-dimensional aggregates. *Proceedings of the 1997 ACM SIGMOD Conference*, pages 159–170, 1997.

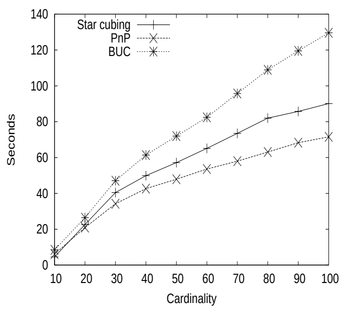

Figure 6. Sequential PnP. Full cube, varying cardinality. Fixed t=1M, d=6, s=0, m=1.

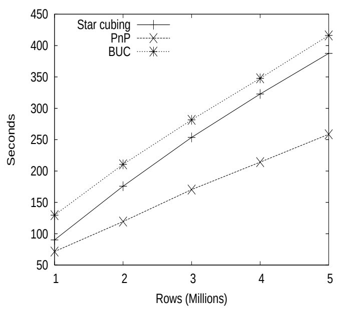

Figure 7. Sequential PnP. Full cube, varying input size. Fixed d=6, c=100, s=0, m=1.

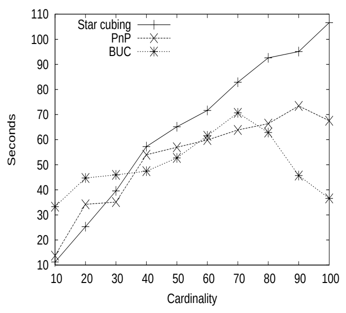

Figure 8. Sequential PnP. Iceberg cube, varying cardinality. Fixed t=5M, d=6, s=0, m=10.

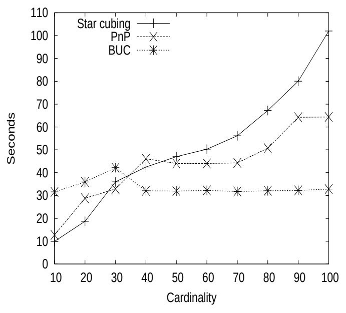

Figure 9. Sequential PnP. Iceberg cube, varying cardinality. Fixed t=5M, d=6, s=0, m=100.

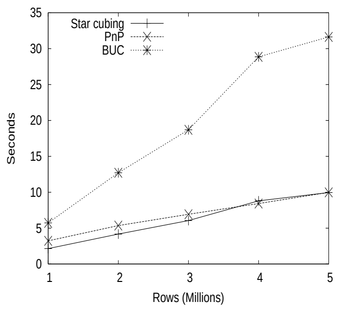

Figure 10. Sequential PnP. Iceberg cube, varying input size. Fixed d=6, c=10, m=100, s=0.

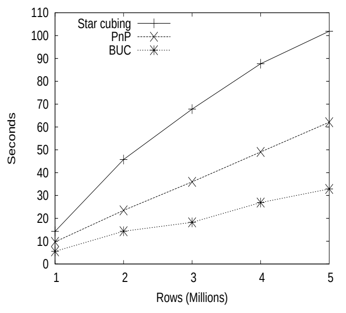

Figure 11. Sequential PnP. Iceberg cube, varying input size. Fixed d=6, c=100, m=100, s=0.

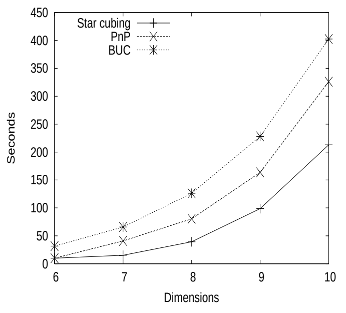

Figure 12. Sequential PnP. Iceberg cube, varying dimensionality. Fixed t=5M, c=10, m=100, s=0.

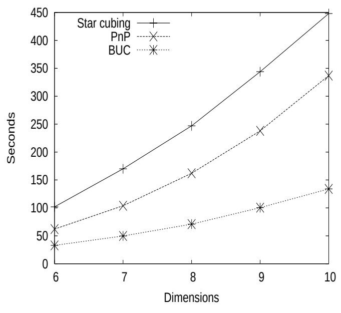

Figure 13. Sequential PnP. Iceberg cube, varying dimensionality. Fixed t=5M, c=100, m=100, s=0.

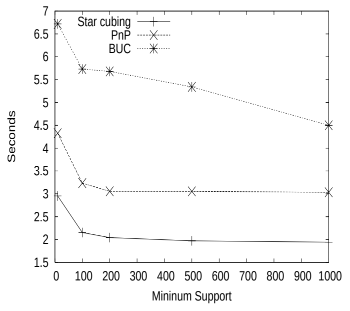

Figure 14. Sequential PnP. Iceberg cube, varying min support. Fixed t=1M, d=6, c=10, s=0.

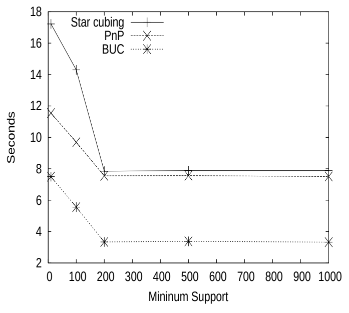

Figure 15. Sequential PnP. Iceberg cube, varying min support. Fixed t=1M, d=6, c=100, s=0.

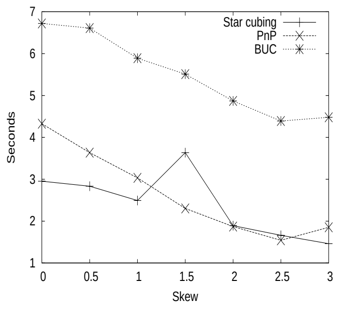

Figure 16. Sequential PnP. Iceberg cube, varying skew. Fixed t=1M, d=6, c=10, m=10.

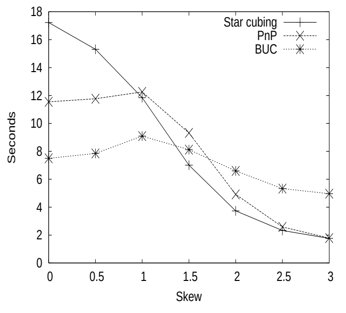

Figure 17. Sequential PnP. Iceberg cube, varying skew. Fixed t=1M, d=6, c=100, m=10.

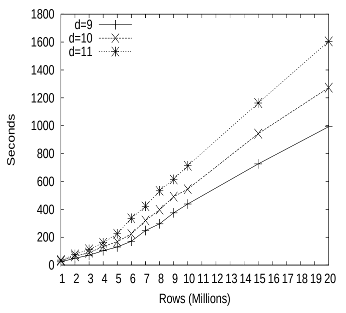

Figure 18. External Memory PnP. Varying dimensionality. Fixed c=300, m=1000, s=0, b=500M.

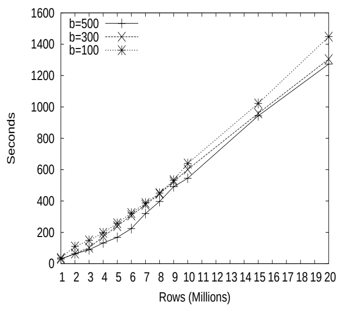

Figure 19. External Memory PnP. Varying input size. Fixed d=10, c=300, m=1000, s=0.

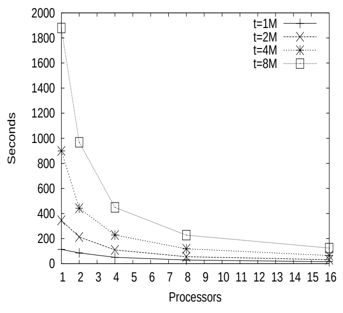

Figure 20. Parallel PnP. Varying input size. Fixed d=10, c=100, m=100, s=0.

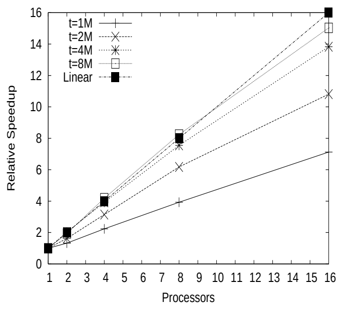

Figure 21. Parallel PnP. Speedup of Figure 20. Fixed d=10, c=100, m=100, s=0.

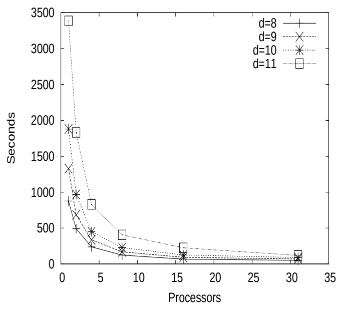

Figure 22. Parallel PnP. Varying dimensions. Fixed t=8M, c=100, m=100, s=0.

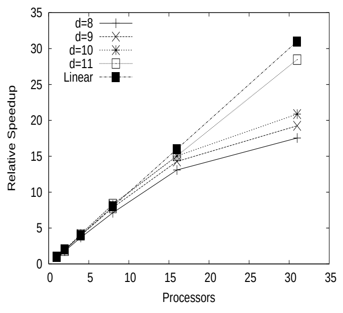

Figure 23. Parallel PnP. Speedup of Figure 22. Fixed t=8M, c=100, m=100, s=0.

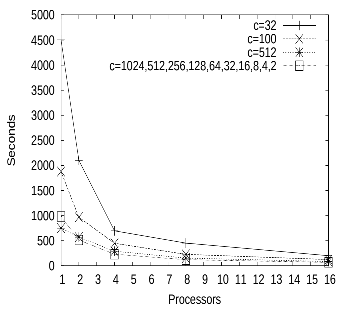

Figure 24. Parallel PnP. Varying cardinality. Fixed t=8M, d=10, m=100, s=0.

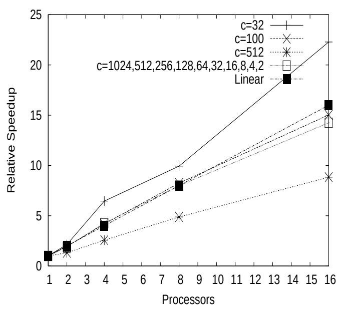

Figure 25. Parallel PnP. Speedup of Figure 24. Fixed t=8M, d=10, m=100, s=0.

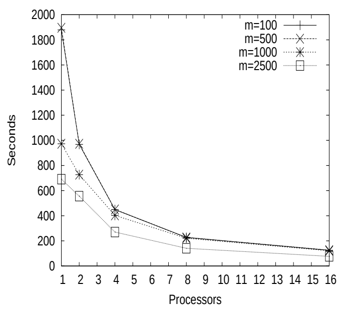

Figure 26. Parallel PnP. Varying minimum support. Fixed t=8M, d=10, c=100, s=0.

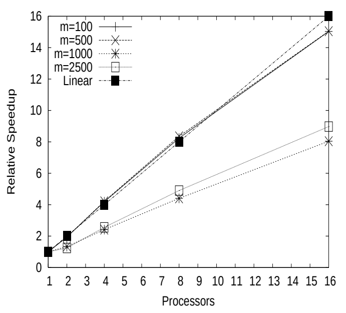

Figure 27. Parallel PnP. Speedup of Figure 26. Fixed t=8M, d=10, c=100, s=0.

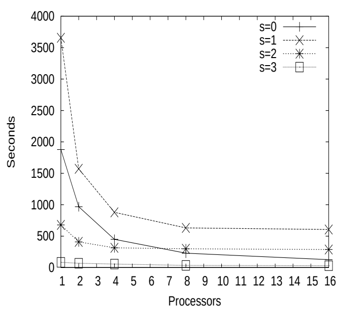

Figure 28. Parallel PnP. Varying skew. Fixed t=8M, d=10, c=100, m=100.

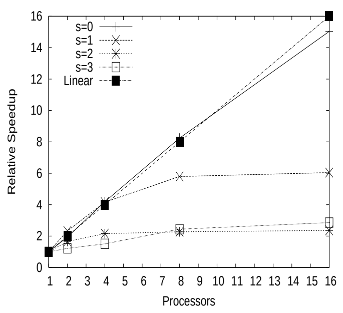

Figure 29. Parallel PnP. Speedup of Figure 28. Fixed t=8M, d=10, c=100, m=100.