İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF INFORMATICS

AN EMPIRICAL STUDY: PARALLEL EXPERIMENTS ON POSTGRESQL

M.Sc. Thesis by
Reydan ÇANKUR

Department : Computational Science and Engineering

Program : Computational Science and Engineering

Thesis Supervisor: Prof. Dr. H. Nüzhet DALFES

MAY 2010

İSTANBUL TECHNICAL UNIVERSITY ★ INSTITUTE OF INFORMATICS

AN EMPIRICAL STUDY: PARALLEL EXPERIMENTS ON POSTGRESQL

M.Sc. Thesis by
Reydan ÇANKUR
(702071014)

Date of submission : 07 May 2010
Date of defence examination: 10 June 2010

Supervisor (Chairman) : Prof. Dr. H. Nüzhet DALFES (ITU)
Members of the Examining Committee : Prof. Dr. M. Serdar ÇELEBİ (ITU)
Prof. Dr. Hasan DAĞ (ITU)

MAY 2010

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ BİLİŞİM ENSTİTÜSÜ

POSTGRESQL VERİTABANI ÜZERİNDE PARALEL DENEMELER

YÜKSEK LİSANS TEZİ
Reydan ÇANKUR
(702071014)

Tezin Enstitüye Verildiği Tarih : 07 Mayıs 2010
Tezin Savunulduğu Tarih : 10 Haziran 2010

Tez Danışmanı : Prof. Dr. H. Nüzhet DALFES (İTÜ)
Diğer Jüri Üyeleri : Prof. Dr. M. Serdar ÇELEBİ (İTÜ)
Prof. Dr. Hasan DAĞ (İTÜ)

MAYIS 2010

FOREWORD

The present thesis was prepared in the Computational Science and Engineering program at the Informatics Institute of İstanbul Technical University (ITU), Turkey from June 2009 to May 2010.

I would like to express my deep appreciation and thanks for my advisor, Prof. Dr. H. Nüzhet DALFES for giving me the chance to do the present work, and for his advices and suggestions.

I would like to thank TUBİTAK, BİDEB for supporting the present thesis.

Computational part of the study was carried out at National Center for High Performance Computing (NCHPC), I would like to thank all people in NCHPC.

Finally, I would like to thank my family and Ufuk YAŞAR.

May 2010

Reydan Çankur

Computational Science and Engineering

TABLE OF CONTENTS

Page

ABBREVIATIONS

TPS	**:** Transaction per Second
App	**:** Appendix
MPP	**:** Massively Parallel Processors
GPL	**:** General Public License
TPC	**:** Transaction Processing Performance Council
TPC-B	**:** Transaction Processing Performance Council - Transactions per Second Benchmark
OpenMP	**:** Open Multi-Processing
API	**:** Application Programming Interface

LIST OF TABLES

Page

LIST OF FIGURES

Page

PARALLEL EXPERIMENTS ON POSTGRESQL

SUMMARY

Database management systems are increasingly used for developing solutions in data-intensive applications. Furthermore, as databases are growing in size and queries increasing in complexity, performance is becoming an issue. Parallel databases can provide speedup and scale up during query processing which is the key solution for handling complex and large databases effectively [1].

In this study our aim is to assess the state of art in parallel relational databases and related performance issues. To achieve our goal, we have concentrated on *PostgreSQL* [2] performance measurement and query optimization by using *OpenMP* [3]. We first explore the performance issues of *PostgreSQL* by profiling the database with *Gprof* [4] and *OProfile* [5]. A query set, which consists of Select, Sort, Group, Inner Join Operations and Aggregate functions, is used to measure the performance on the stated profilers. In addition to that, *pgbench* [6], which is a *PostgreSQL* benchmarking tool, is used to evaluate the database performance on different CPU-core numbers, in terms of transaction per second (TPS) and CPU time. By obtaining data on which functions the most time during query processing is spent, we try to find out where to implement multi-processing. The results show that implementation of *OpenMP* to the source code increases the performance for some aspects.

POSTGRESQL VERİTABANI ÜZERİNDE PARALEL DENEMELER

ÖZET

Veritabanı yönetim sistemleri, veri yoğun uygulamalarda çözümler geliştirmek için kullanılır. Günümüzde hem akademide hem de endüstride kullanılan veritabanlarının boyutu ve sorguların karmaşıklığı artmaktadır. Boyutları artan veritabanları için performans bir sorun haline gelmektedir. Bu noktada paralel veritabanları hız sağlayarak veritabanlarının etkin kullanım için anahtar çözüm olmaktadır.

Bu çalışmada amacımız paralel ilişkisel veritabanları ve ilgili performans sorunlarını değerlendirmektir. Hedefimizi gerçekleştirmek için, *PostgreSQL* veritabanının performans ölçümü ve sorgu iyileştirmesine yöneldik. Sorgu iyileştirmeleri için *OpenMP* dili kullanıldı. İlk olarak çeşitli yollarla varolan kaynak kodun performans ölçümleri alındı. Bunu yapmak için çeşitli görevler içeren bir sorgu seti veritabanı üzerinde çalıştırıldı. Daha sonra en çok zaman harcayan fonksiyonlar belirlenerek, *OpenMP* dili ile tekrar yazıldı.

Sonuçlar bize bu yöntemle performansın arttırılabileceğini gösteriyor.

1. INTRODUCTION

Database management systems are increasingly used for developing solutions in data-intensive applications. Data storage and retrieval is essential for all kinds of applications. A database management system is the key factor to manage the organized collection of data, also with database management system's data can be stored, queried and reports can be produced. Information management gains more importance in this era therefore it is not surprising for databases to grow to huge sizes and be accessed by multiple users. Furthermore, queries and reports to be retrieved from databases increase in complexity.

Many applications today require more computing power than a sequential computer can offer. Parallel processing provides solution to this problem by increasing the number of computing elements such as cores, in a computer.

Parallel computing is a computer science discipline that deals with the concurrent execution of applications. It has been decades that parallel computing is a research area but now it is emerging due to computer industry's shift to multi-core processors.

The development of parallel processing is influenced by many factors.

Computational requirements are increasing both in the area of scientific and business applications. Data mining, telecommunications, climate modelling, and, car crash simulation are the significant areas that are in need of more computational power and speed.

The physical limitation of serial computers and the current trend in multi-core computers are also the essential motivations for parallel computing.

The interest in parallel computing dates back to the late 1950's. In 60's and 70's there were shared memory multiprocessors, with multiple processors working side-by-side on shared data. In the mid 1980's, massively parallel processors (MPPs) came to dominate the top end of computing. Starting in the late 80's, clusters came to compete and displace MPPs for many applications. Moreover, today, parallel computing is becoming mainstream based on multi-core processors. The aim is to increase overall processing performance by adding additional cores to CPU [7].

Nowadays, the daily volumes of data being added to some databases are measured in terabytes. Furthermore, as databases are growing in size and queries increasing in complexity, performance is becoming an issue.

At that point parallel database processing becomes an alternative solution. Parallel databases can provide speedup and scale up during query processing which is the key solution for handling complex and large databases effectively [1]. The driving force behind parallel database processing includes; querying large databases, increasing availability of the system, and processing large number of transactions per second [8].

There are many database management softwares both commercial and open source. Some popular examples can be found below.

Microsoft SQL Server designed to create web, enterprise, and desktop database systems. It is used with various goals and at different levels. MS SQL Server allows you to store large amount of data, which handles components like video, photographs, numbers, text, and much more. *Microsoft SQL Server* is developed to manage terra bytes of data in comparison with *Microsoft Access* that can handle only one gigabyte of data.

Oracle is one of the leading commercial SQL relational database management systems. It is available in a variety of configurations from small personal versions to fail-safe, enterprise-class versions. *Oracle* offers lots of features and functionality for solving complicated problems of medium and large enterprise business applications and warehouses. This powerful system requires deep knowledge and skill to handle large environments.

MySQL runs as a service providing multiple user access to several databases. *MySQL* is popular for web applications and operates with the database elements for the platforms (Linux/BSD/Mac/Windows). *MySQL* popularity for use with web applications is closely associated to the popularity of PHP programming language which is often used along with *MySQL*. Many high-traffic web sites use *MySQL* as the backend for its data warehouse. *MySQL* is very popular with start up companies, small or medium businesses and projects because it is easy to use at a low cost. In case when high speed reads are applied for web, gaming and medium or small data storages *MySQL* surpasses all the other database management systems [9].

PostgreSQL is a relational DBMS that many web application developers prefer as the back-end data management component. It's principally used by many distinguished organizations applying it for mission critical or wide-ranging applications. The .info and .org domain name registries its use as their primary data store, so do many financial institutions and large companies. Key advantages, such as open source community support, very low deployment cost, and easy administration, make it the great choice for those who use it for database driven website development [9].

We have chosen *PostgreSQL* database to apply parallelism. *PostgreSQL* is one of the most powerful open source databases. It is a premium public domain database, which is object-oriented and supports spatial features. *PostgreSQL* database is written in pure C and the uniformity simplified to implement parallelism. *PostgreSQL* 8.3.9 version is used on all steps of the study.

1.1 Purpose of the Thesis

In this study, our aim is to assess the state of art in parallel relational databases and related performance issues. The two main objectives of this study are analyzing the *PostgreSQL* database performance by using parallel sources and make a research on parallelizing time-consuming functions in order to increase overall performance.

2. ANALYZING POSTGRESQL DATABASE ENGINE

2.1 Objectives

In this section, we have analyzed the performance of *PostgreSQL* database. The objective of performance analysis is to understand the database engine, time-consuming functions and to use the information gained during parallelization. We have created a set of queries that perform select, aggregate and join tasks. After query preparation, we run these queries with different optimization levels of ICC compiler. We have also profiled the database engine with two different profilers; *GNU gprof* and *OProfile*.

We have run all tests and benchmarks on a HP Integrity Superdome server with 32 Intel Itanium2 processors running at 1.6 GHz (dual-core). The memory of this server is 128 GB.

Figure 2.1 explains the overall process of the work done. Below are the results and details.

2.2 Profiling *PostgreSQL*

Profiling provides information about where the program spent its time, and which functions call the other functions while executing. This information can show which pieces of the program are slower than expected, and might be candidates for rewriting to make the program execute faster. It can also tell which functions are being called how many times.

Since the profiler uses information collected data during the actual execution of programs, it can be used on programs that are too large or too complex to analyze by reading the source.

Profiling has several steps:

- Compilation with profiling option enabled.
- Execution of the program to generate a profile data file.
- Analysis of the profile data.

2.3 Profiling Tools

We have profiled *PostgreSQL* database engine with two different profilers and we have made some performance tests with *PostgreSQL*'s internal tool EXPLAIN ANALYZE. The profilers are *GNU gprof* and *OProfile*.

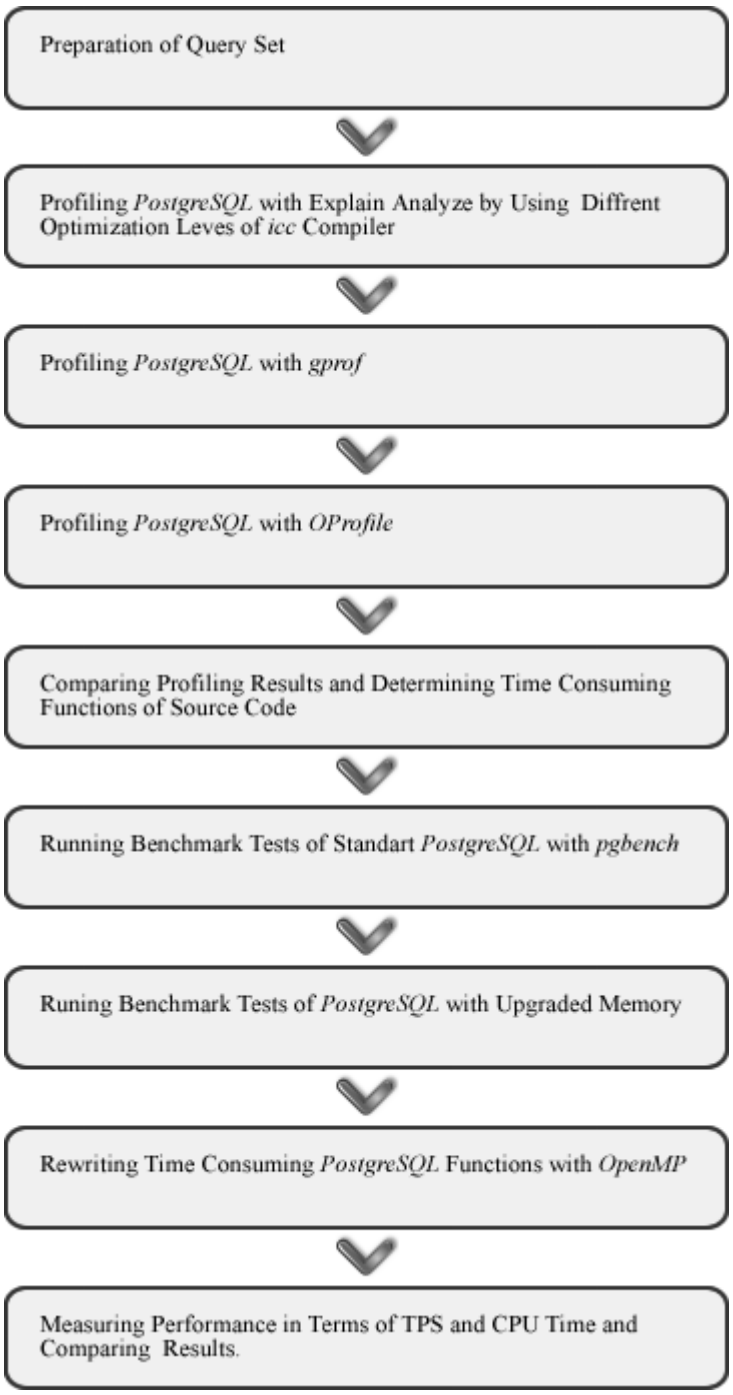


Figure 2.1 : Flow Chart of Overall Process.

2.3.1 GNU gprof

The *gprof* utility produces an execution profile of C, Pascal, or Fortran77 programs. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file which is created by programs that are compiled with the -pg option. The given object file establishes the relation between its symbol table and the call graph profile. The default graph profile file name is the name of the executable with the suffix .gmon appended. If more than one profile file is specified, the *gprof* output shows the sum of the profile information in the given profile files. The *gprof* utility calculates the amount of time spent in each routine. Next, these times are propagated along the edges of the call graph. The functions are displayed sorted according to the time they represent including the time of their call graph descendants. And each function entry is shown its (direct) call graph children, and how their times are propagated to this function.

A similar display of function shows how this function's time and the time of its descendants are propagated to its (direct) call graph parents. Second, a flat profile is given. This listing gives the total execution times, the call counts, the time in msec or usec the call spent in the routine itself, and the time in msec or usec the call spent in the routine itself including its descendants [4].

2.3.2 OProfile

OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. *OProfile* is released under the GNU GPL. It consists of a kernel driver and a daemon for collecting sample data, and several post-profiling tools for turning data into information. *OProfile* leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

The early versions of *OProfile* were developed as part credit for a M.Sc. in Computer Science. Compaq’s DCPI profiler inspired the basic principles of the design [5].

2.3.3 Explain Analyze

EXPLAIN command displays the execution plan that the *PostgreSQL* planner generates for the supplied statement. The execution plan shows how the table(s) referenced by the statement will be scanned—by plain sequential scan, index scan, etc.—and if multiple tables are referenced, what join algorithms will be used to bring together the required rows from each input table. The most critical part of the display is the estimated statement execution cost, which is the planner's guess at how long it will take to run the statement (measured in units of disk page fetches). Actually two numbers are shown: the start-up time before the first row can be returned, and the total time to return all the rows.

The ANALYZE option causes the statement to be actually executed, not only planned. The total elapsed time expended within each plan node (in milliseconds) and total number of rows it actually returned is added to the display. This is useful for seeing whether the planner's estimates are close to reality [2].

We have run EXPLAIN ANALYZE command with a version that is compiled by using ICC with different optimization levels and query performance is evaluated with all six optimization levels.

2.4 Query Sets

We have prepared a query set which consists of select, insert, aggregate and join tasks. Profiling tasks are completed during query set execution. There are 17 different queries within the set. We have run these queries on a five million record table, which is created with below queries. We have created two tables with same CREATE script in order to perform join tasks.

Query 1:

```
CREATE TABLE strint1 AS
SELECT sid,
       md5((sid*10)::text),
       ((SUBSTRING(RANDOM()::text FROM 3 for 5))::int+10000) AS
      string
FROM generate_series(1,1000000) sid;
```

Query 2:

```
INSERT INTO strint1
SELECT sid,
       md5((sid*99)::text),
       ((SUBSTRING(RANDOM()::text FROM 3 for 5))::int+10000) AS
string
FROM   generate_series(1000001,5000000) sid;
```

We have evaluated below benchmark tasks;

2.4.1 Selection Task

Search operation in databases is accomplished by selection operations. Selection is a process that selects a set of records based on a filter from a given table. Figure 2.2 gives a graphical representation of selection operation. In SQL, selection operation is implemented by using WHERE clause where the filter is applied. Below you can find selection queries that we have prepared for benchmarking [8].

2.4.1.1 Search without Filter

Select operation is completed without any filter. Query retrieves all the records from the given table. An example query that we have run can be seen below.

Query 3:

```
SELECT md5
FROM   strint1;
```

2.4.1.2 Exact-Match Search

An exact match search query is a query that selection process tries to find an exact match between the filter criteria and the input table. Below is the query that we have performed for exact match search.

Query 4:

```
SELECT sid,
       md5
FROM   strint1
WHERE  sid = 3008;
```

Figure 2.2 : Select with Filter.

2.4.1.3 Range Query Search

A range query search is a query that retrieves a continuous range from the given table. Greater than (>) or less than (<) operators can be used to retrieve the range.

Query 5:

```
SELECT string,
       md5
FROM   strint1
WHERE  string > 60000;
```

2.4.2 Sorting Task

Sorting is an operation that arranges the records in a particular order depending on one or more attributes. Sorting queries can order the records ascending and descending. We have performed both types of sorting queries. The result table is ordered as ascending by default.

Query 6:

```
SELECT *
FROM   strint1
ORDER BY
       string;
```

Query 7:

```
SELECT *
FROM   strint1
ORDER BY
       md5 DESC;
```

2.4.3 Duplicate Removal Task

Duplicate removal is closely associated with sorting. In SQL, Distinct operator in Select clause carries out the duplicate removal operation. Distinct removes all duplicates from the result table.

Query 8:

```
SELECT DISTINCT
       string
FROM   strint1;
```

2.4.4 Queries with Aggregate Functions

Aggregate functions perform a calculation on a set of values and return a single value. Except for COUNT, aggregate functions ignore null values.

All aggregate functions are deterministic which means aggregate functions return the same value any time that they are called by using a specific set of input values.

Useful aggregate functions:

- AVG() - Returns the average value
- COUNT() - Returns the number of rows

- FIRST() - Returns the first value
- LAST() - Returns the last value
- MAX() - Returns the largest value
- MIN() - Returns the smallest value
- SUM() - Returns the sum

Query 9:

```
SELECT AVG(string)
FROM   strint1;
```

Query 10:

```
SELECT MIN(string)
FROM   strint1;
```

Query 11:

```
SELECT MAX(string)
FROM   strint1;
```

Query 12:

```
SELECT SUM(string)
FROM   strint1;
```

Query 13:

```
SELECT COUNT(*)
FROM   strint1;
```

2.4.5 Group By Task

The SQL GROUP BY statement is used together with the SQL aggregate functions like SUM or COUNT to group the retrieved data by one or more columns.

Query 14:

```
SELECT string,
       COUNT(*)
FROM   strint1
GROUP BY
       string;
```

2.4.6 Join Task

Information is divided into multiple tables in relational databases as a result of normalization. Consequently, if the information needs to be combined in order to be displayed as a report, data shall be retrieved from multiple tables through join operation. Join operation is one of the most complex operations in a database. GROUP BY clause can be used to group data from multiple tables and HAVING can be used to implement a filter to the result table.

Query 15:

```
SELECT s1.sid,
       s2.sid,
       s1.string,
       s2.string
FROM   strint1 s1
       INNER JOIN strint2 s2 ON s1.sid=s2.sid
WHERE  s1.md5 = s2.md5;
```

Query 16:

```
SELECT s1.string,
       COUNT(*)
FROM   strint1 s1
       INNER JOIN strint2 s2 ON s1.md5=s2.md5
GROUP BY
       s1.string;
```

Query 17:

```
SELECT s1.string,
       COUNT(*)
FROM   strint1 s1
       INNER JOIN strint2 s2 ON s1.md5=s2.md5
GROUP BY
       s1.string
HAVING s1.string > 58000;
```

2.5 Profiling Results

2.5.1 ICC Optimization Results with EXPLAIN ANALYZE

First, we have compiled *PostgreSQL* source code with different optimization levels of ICC in order to see the effects. Below you can find chart of the results represented with milliseconds for every optimization level.

Table 2.1: Query execution duration for different optimization levels.

Query Set	Optimization Levels					
	O0	O1	O2	O3	O4	O5
3-Search without Filter	10939.491	3670.350	3871.032	3921.139	3840.643	3827.296
4-Exact-Match Search	7392.407	2097.358	2366.327	2163.935	2241.868	2274.762
5-Range Query Search	12562.777	3791.762	3946.048	3869.790	4007.645	3997.288
6-Sorting Ascending	136071.030	35881.136	42921.548	45068.325	44082.452	43374.132
7-Sorting Descending	241653.153	110879.043	114574.546	116340.130	113646.134	114055.311
8-Duplicate Removal	125669.221	26636.845	26649.313	26675.155	26704.750	27629.638
9-Aggregate Func.-Avg	12293.752	3789.910	4359.607	4197.788	4398.691	4363.671
10-Aggregate Func.-Min	11940.819	3878.251	4066.790	3865.553	3946.364	3968.833
11-Aggregate Func.-Max	11905.979	3886.842	4023.895	3965.621	3922.074	3969.906
12-Aggregate Func.-Sum	11559.718	3712.478	4217.570	3835.926	3887.419	3922.566
13-Aggregate Func.-Count	7799.731	2980.971	3173.311	3216.890	3118.850	3126.296
14-Group By	127672.987	27102.379	27625.140	27520.285	27689.036	27401.387
15-Join with Filter	92910.226	34629.468	39594.079	36838.298	33492.973	40856.356
16-Join with Group By	109303.839	40732.481	43009.285	40198.491	38064.650	40892.929
17-Join with Having	77250.985	29980.541	27354.002	27367.625	29286.631	27444.773

As seen on table 2.1 the best performance has been seen when the source code is compiled with –O3 option. Therefore, we made all profiling and benchmarking on a database that is compiled with –O3 option.

2.5.2 *Gprof* Results

Table 2.2: *gprof* results for search without filter query.

order	each time	cumulative seconds	self seconds	Calls	s/call	s/call	name
1	6.81	0.14	0.14	15000516	0	0	AllocSetFree
2	6.71	0.29	0.14	15001947	0	0	AllocSetAlloc
3	*5.34*	*0.4*	*0.11*	*5000000*	*0*	*0*	*ExecProject*
4	4.82	0.5	0.1	5000001	0	0	heapgettup_pagemode
5	4.78	0.6	0.1	1	0.1	2.08	standard_ExecutorRun
6	4.59	0.7	0.1	5000000	0	0	printtup
7	*3.97*	*0.78*	*0.08*	*5000000*	*0*	*0*	*slot_deform_tuple*
8	3.36	0.85	0.07	15000048	0	0	internal_putbytes
9	**3.36**	**0.93**	**0.07**	**5000001**	**0**	**0**	**ExecScan**
10	*3.22*	*0.99*	*0.07*	*5000001*	*0*	*0*	*ExecProcNode*
…	…	…	…	…	…	…	…
24	*1.51*	*1.65*	*0.03*	*5000008*	*0*	*0*	*ExecClearTuple*
…	…	…	…	…	…	…	…
40	**0.66**	**2**	**0.01**	**5000000**	**0**	**0**	**ExecStoreTuple**

Table 2.3: *gprof* results for exact-match search query.

order	each time	cumulative seconds	self seconds	calls	s/call	s/call	name
1	**13.41**	**0.09**	**0.09**	**4999999**	**0**	**0**	**ExecMakeFunctionResultNoSets**
2	*8.94*	*0.15*	*0.06*	*5000001*	*0*	*0*	*slot_deform_tuple*
3	7	0.2	0.05	5000001	0	0	heapgettup_pagemode
4	6.71	0.24	0.04	5000000	0	0	slot_getattr
5	*5.96*	*0.28*	*0.04*	*5000000*	*0*	*0*	*ExecQual*
6	4.92	0.32	0.03	5000001	0	0	heap_getnext
7	*4.92*	*0.35*	*0.03*	*4999999*	*0*	*0*	*ExecEvalScalarVar*
8	4.62	0.38	0.03	5000001	0	0	SeqNext
9	4.32	0.41	0.03	5000003	0	0	AllocSetReset
10	4.32	0.44	0.03	5000000	0	0	HeapTupleSatisfiesMVCC
11	**3.87**	**0.46**	**0.03**	**5000000**	**0**	**0**	**ExecStoreTuple**
12	2.83	0.48	0.02	5000060	0	0	check_stack_depth
13	2.83	0.5	0.02	46729	0	0	heapgetpage
14	*2.68*	*0.52*	*0.02*	*5000000*	*0*	*0*	*ExecEvalConst*
15	2.53	0.54	0.02	5000000	0	0	int4eq
16	**2.38**	**0.55**	**0.02**	**2**	**8**	**318.45**	**ExecScan**
17	2.24	0.57	0.01	5000001	0	0	MemoryContextReset
18	1.94	0.58	0.01	5000000	0	0	pgstat_init_function_usage
19	1.79	0.59	0.01	272400	0	0	transtime
20	1.64	0.6	0.01	5000006	0	0	TransactionIdPrecedes

Table 2.4: *gprof* results for insert query.

order	each time	cumulative seconds	self seconds	Calls	s/call	s/call	name
1	11.42	1.18	1.18	4000000	0	0	pg_md5_hash
2	8.28	2.03	0.85	4000001	0	0	XLogInsert
3	**5.92**	**2.65**	**0.61**	**19999995**	**0**	**0**	**ExecMakeFunctionResultNoSets**
4	4.78	3.14	0.49	40040211	0	0	AllocSetAlloc
5	3.42	3.49	0.35	95999234	0	0	pg_utf_mblen
6	3.41	3.84	0.35	95999068	0	0	pg_mblen
7	*2.66*	*4.12*	*0.28*	*4000000*	*0*	*0*	*ExecProject*
…	…	…	…	…	…	…	…
15	*1.3*	*5.66*	*0.13*	*16000002*	*0*	*0*	*ExecEvalConst*
16	*1.27*	*5.79*	*0.13*	*12000000*	*0*	*0*	*ExecEvalCoerceViaIO*
…	…	…	…	…	…	…	…
21	*1.04*	*6.38*	*0.11*	*4000001*	*0*	*0*	*ExecProcNode*
22	1.01	6.49	0.1	8000011	0	0	heap_compute_data_size
23	0.98	6.59	0.1	4000000	0	0	RelationGetBufferForTuple
24	0.92	6.68	0.1	4000011	0	0	heap_form_tuple
25	**0.92**	**6.78**	**0.1**	**4000001**	**0**	**0**	**ExecScan**
…	…	…	…	…	…	…	…
46	*0.53*	*8.28*	*0.06*	*4000000*	*0*	*0*	*slot_deform_tuple*
…	…	…	…	…	…	…	…
64	*0.35*	*9.08*	*0.04*	*4000009*	*0*	*0*	*ExecClearTuple*
…	…	…	…	…	…	…	…
67	*0.35*	*9.19*	*0.04*	*3999999*	*0*	*0*	*ExecEvalScalarVar*

Profiling results of *gprof* are evaluated for all queries within the query set but only SELECT and INSERT query results are displayed here. The benchmarking query that we have run comprises Select, Insert and Update operations therefore we have determined time consuming functions regarding these queries' profiling results. Lines that are highlighted with bold represents the functions that we have successfully parallelized, italic functions symbolizes the ones that are tried to be parallelized but failed.

2.5.2.1 Call Graphs

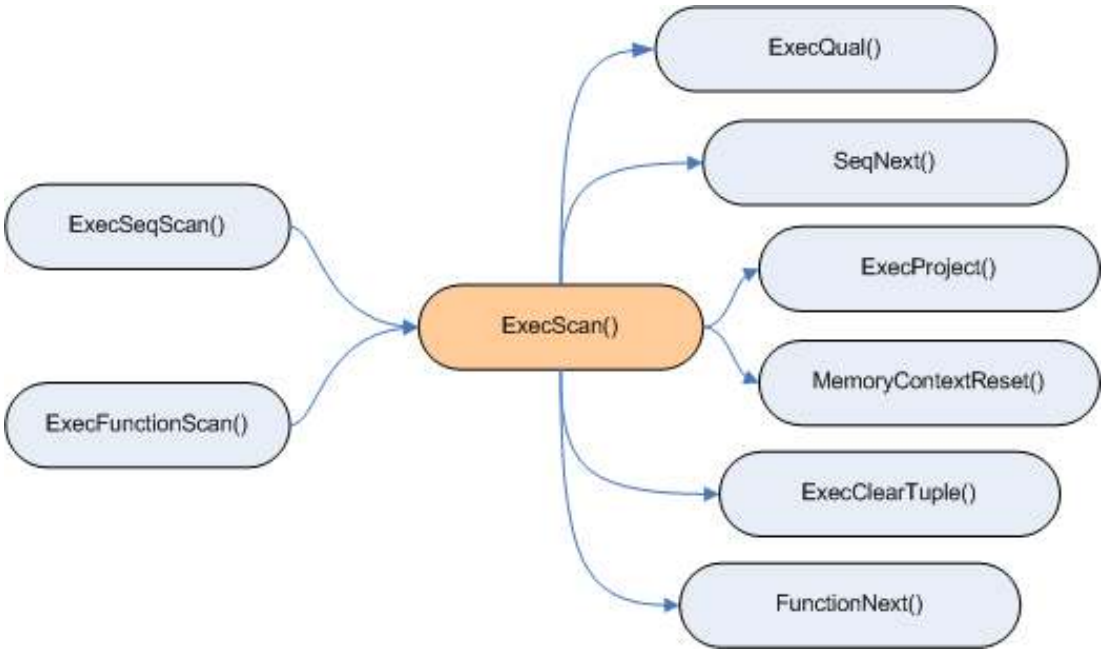


Figure 2.3 : Call Graph of Function ExecScan.

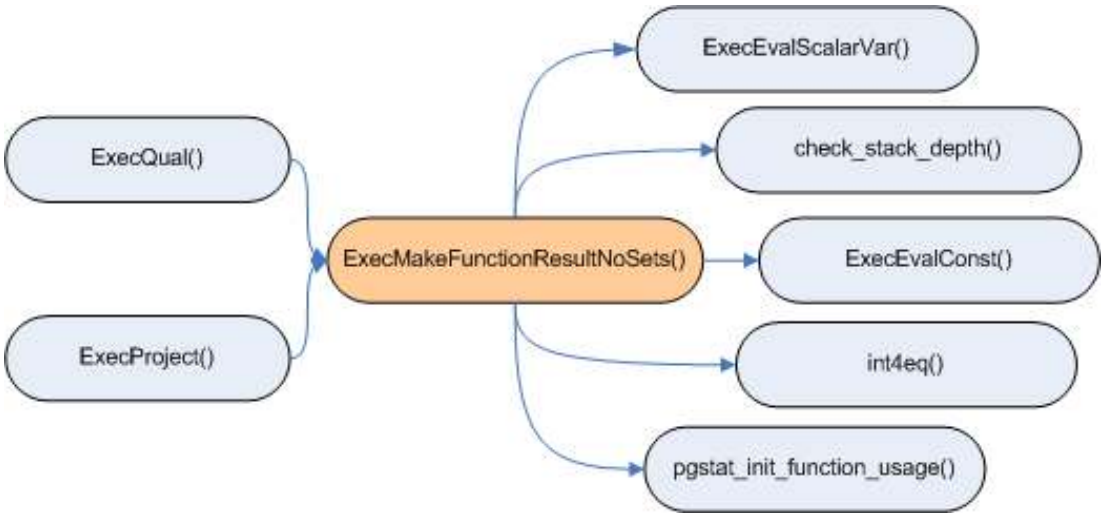


Figure 2.4 : Call Graph of Function ExecMakeFunctionResultNoSets.

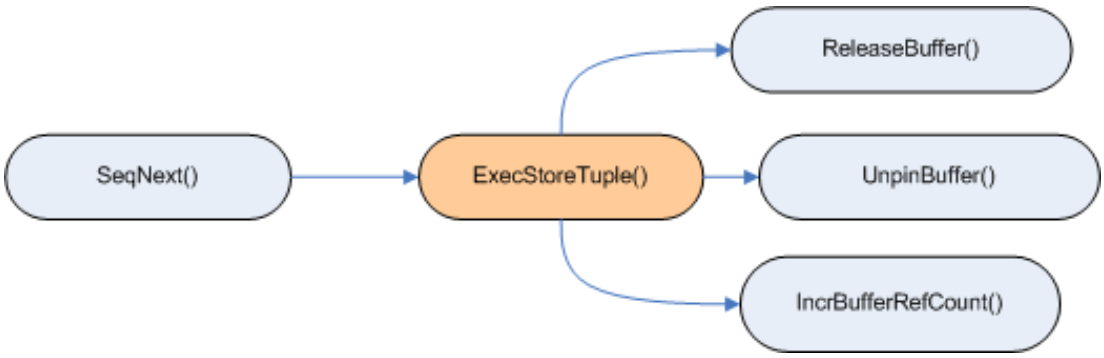


Figure 2.5 : Call Graph of Function ExecStoreTuple.

2.5.3 *OProfile* Results

Table 2.5: *OProfile* results for search without filter query.

Order	Samples	%	Symbol Name
1	*2489*	*9.7923*	*AllocSetFreeIndex*
2	*1798*	*7.0737*	*slot_deform_tuple*
3	1327	5.2207	.plt
4	1135	4.4653	internal_putbytes
5	*1080*	*4.2490*	*ExecProject*
6	1067	4.1978	AllocSetAlloc
7	892	3.5093	AllocSetFree
8	882	3.4700	appendBinaryStringInfo
9	875	3.4424	printtup
10	834	3.2811	heapgettup_pagemode
11	*647*	*2.5454*	*ExecProcNode*
12	626	2.4628	enlargeStringInfo
13	**607**	**2.3881**	**ExecScan**
14	598	2.3527	pq_sendint
15	595	2.3409	heap_getnext
16	579	2.2779	text_to_cstring
17	*566*	*2.2268*	*ExecutePlan*
18	521	2.0497	slot_getsomeattrs
19	467	1.8373	pfree
20	439	1.7271	heapgetpage
21	399	1.5698	SeqNext
22	374	1.4714	MemoryContextAlloc
23	**340**	**1.3376**	**ExecSelect**
24	337	1.3258	HeapTupleSatisfiesMVCC
25	336	1.3219	heap_tuple_untoast_attr
26	312	1.2275	FunctionCall1
27	289	1.1370	TransactionIdPrecedes
28	288	1.1331	pq_putmessage
29	**277**	**1.0898**	**ExecStoreTuple**
30	*275*	*1.0819*	*ExecClearTuple*

Table 2.6: *OProfile* results for exact-match search query.

Order	Samples	%	Symbol Name
1	**1082**	**12.3799**	**ExecMakeFunctionResultNoSets**
2	*1037*	*11.865*	*slot_deform_tuple*
3	720	8.238	heapgettup_pagemode
4	545	6.2357	slot_getattr
5	428	4.897	heapgetpage
6	407	4.6568	SeqNext
7	396	4.5309	heap_getnext
8	377	4.3135	.plt
9	*364*	*4.1648*	*ExecEvalScalarVar*
10	322	3.6842	list_head
11	*314*	*3.5927*	*ExecQual*
12	**248**	**2.8375**	**ExecStoreTuple**
13	242	2.7689	HeapTupleSatisfiesMVCC
14	237	2.7117	MemoryContextSwitchTo
15	**220**	**2.5172**	**ExecScan**
16	216	2.4714	TransactionIdPrecedes
17	*211*	*2.4142*	*ExecEvalConst*
18	206	2.357	int4eq
19	186	2.1281	MemoryContextReset
20	181	2.0709	check_stack_depth

Table 2.7: *OProfile* results for insert query.

Order	Samples	%	Symbol Name
1	16401	10.9567	XLogInsert
2	8793	5.8742	calculateDigestFromBuffer
3	8572	5.7265	doTheRounds
4	**6513**	**4.351**	**ExecMakeFunctionResultNoSets**
…	…	…	…
9	*3792*	*2.5333*	*AllocSetFreeIndex*
…	…	…	…
17	*1835*	*1.2259*	*ExecTargetList*
…	…	…	…
22	*1586*	*1.0595*	*ExecEvalCoerceViaIO*
…	…	…	…
26	*1284*	*0.8578*	*ExecProject*
…	…	…	…
30	*1162*	*0.7763*	*slot_deform_tuple*
31	1098	0.7335	pgstat_init_function_usage
32	*1066*	*0.7121*	*ExecEvalConst*
…	…	…	…
43	**798**	**0.5331**	**ExecScan**
44	760	0.5077	pgstat_end_function_usage
45	*754*	*0.5037*	*ExecutePlan*
46	*694*	*0.4636*	*ExecProcNode*
47	*647*	*0.4322*	*ExecInsert*
…	…	…	…
65	*475*	*0.3173*	*ExecEvalScalarVar*
…	…	…	…
85	*369*	*0.2465*	*ExecClearTuple*
…	…	…	…
142	**60**	**0.0401**	**ExecSelect**

Profiling results for *OProfile* are evaluated for all queries within the query set but only SELECT and INSERT query results are displayed here. The benchmarking query that we have run comprises Select, Insert and Update operations therefore we have determined time consuming functions regarding these queries' profiling results. Lines that are highlighted with bold represents the functions that we have successfully parallelized, italic functions symbolizes the ones that are tried to be parallelized but failed.

2.6 Performance Analyses

Benchmarking is the process of comparing one's business processes and performance metrics to industry bests and/or best practices from other industries. The term benchmarking was first used by cobblers to measure people's feet for shoes. They would place someone's foot on a "bench" and mark it out to make the pattern for the shoes. Benchmarking is most used to measure performance using a specific indicator (cost per unit of measure, productivity per unit of measure, cycle time of x per unit of measure or defects per unit of measure) resulting in a metric of performance that is then compared to others [10].

In computing, benchmark is the work of running a computer program or a set of operations in order to evaluate the relative performance of an object. This evaluation can be achieved by running standard tests and trials. Benchmarking is usually associated with assessing performance characteristics of computer hardware, for example, the floating point operation performance of a CPU, but there are circumstances when the technique is also applicable to software. Software benchmarks are, for example, run against compilers or database management systems [11].

2.6.1 Reasons for Database Benchmarking

Benchmarks are performed for various reasons. However, benchmarks are primarily used:

- To compare different hardware configurations

  Benchmarks can be used to compare the relative performance of different hardware running the same application. This is generally used to directly compare hardware configurations between two hardware vendors.

- To compare different database vendor software

  By running the same benchmark using different database software on the same machine, one can easily compare between different database vendors. This is generally used to make a price/performance decision between vendors such as *Oracle*, *Microsoft*, *IBM*, etc.

- To compare different database software releases

  Similar to the above, one can use different versions of the same vendor's database software to compare the one they want to use or check for performance regressions due to upgrades (i.e. 10g vs. 11g) [12].

In our case, we have used database benchmarking in order to compare the standard version of *PostgreSQL* and the version parallelized with *OpenMP*.

Industry standard benchmarks are generally used by businesses to compare different hardware and software system performance for purchase-related reasons. One of the major industry standard benchmark is Transaction Processing Benchmark (TPC) Benchmark. The TPC is a non-profit corporation which supports a consortium of hardware and database software vendors devoted to defining transaction processing and database-related benchmarks. The primary goal behind TPC benchmarks is the definition of functional requirements, which can be run on any database, regardless of the hardware or operating system. The term transaction, looked at as a computer function, could refer to a set of operations including disk read/writes, operating system calls, or some form of data transfer from one subsystem to another [13].

There are variety of TPC benchmarks focusing on different area like TPC-C, TPC-E, TPC-H and TPC-B.

TPC-C is an on-line transaction processing (OLTP) benchmark. TPC-C simulates a complete computing environment where a population of users executes transactions against an order-entry database.

TPC-E is a new on-line transaction processing (OLTP) workload, which is simulating the transactions of a brokerage firm.

TPC-H is an old ad-hoc, decision support benchmark. The benchmark illustrates decision support systems that examine large volumes of data, execute complex queries and give answers to critical business questions.

Another benchmark from TPC that is TPC-B measures throughput in terms of how many transactions per second a system can perform. TPC-B is not an OLTP benchmark; it can be looked at as a database stress test characterized by;

- Significant disk input/output
- Moderate system and application execution time
- Transaction integrity

TPC Benchmark B is targeted at database management systems (DBMS) batch applications and the back-end database server market segment, either stand-alone or client-server. It can be used to measure how many total simultaneous transactions a system can handle [13]. We have used TPC-B like benchmark to evaluate the performance of *PostgreSQL*. We have utilized *pgbench* tool in order to run TPC-B like benchmarks on *PostgreSQL*. *pgbench* is a simple program for running benchmark tests on *PostgreSQL*. It runs the same sequence of SQL commands repeatedly, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (transactions per second). By default, *pgbench* tests a

scenario that is based on TPC-B, involving five SELECT, UPDATE, and INSERT commands per transaction [13]. Below the sql commands that are run for benchmarking can be found:

```
\set nbranches :scale
\set ntellers 10 * :scale
\set naccounts 100000 * :scale
\setrandom aid 1 :naccounts
\setrandom bid 1 :nbranches
\setrandom tid 1 :ntellers
\setrandom delta -5000 5000
BEGIN;
UPDATE accounts SET abalance = abalance + :delta WHERE aid = :aid;
SELECT abalance FROM accounts WHERE aid = :aid;
UPDATE tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
UPDATE branches SET bbalance = bbalance + :delta WHERE bid = :bid;
INSERT INTO history (tid, bid, aid, delta, mtime) VALUES (:tid,
:bid, :aid, :delta, CURRENT_TIMESTAMP);
END;
```

pgbench output reports, the TPS rate figured with and without counting the time to start database sessions. The TPC-B like transaction test requires specific tables to be set up beforehand., to populate these tables *pgbench* shall be invoked with –I option;

```
pgbench –i dbname
```

pgbench `-i` creates four tables `accounts`, `branches`, `history`, and `tellers`, destroying any existing tables of these names. A scale factor to determine the records of tables can be given.

At the default "scale factor" of 1, the tables initially contain this many rows:

```
table            # of rows
-------------------------
branches         1
tellers          10
accounts         100000
history          0
```

We have given 64 as the “scale factor”, and table details are as follows:

```
table            # of rows
-------------------------
branches         64
tellers          640
accounts         6400000
history          0
```

Scale factor “64” creates a benchmark database with size 968 MB. Figure 2.6 displays the list of parameters that is utilized during benchmark test runs.

Benchmarking Parameters		
Scale	:	64
Benchmark Type	:	tpc-b
Total Transactions	:	64000
Repeat Count	:	7
Number of Clients	:	1,2,4,8,16,32,64

Figure 2.6 : Benchmarking Parameter List.

We have set client parameter to “1 2 4 8 16 32 64” in order to evaluate the behaviour of database against different numbers of clients. Scaling factor is determined regarding the largest client number. The number of rows in the “branches” table will equal the scaling factor, and every transaction updates one randomly chosen "branches" row. If the client number is larger than scaling factor, there will be no actual concurrency and all transactions stack up on the single “branches” row. We have set the scaling factor to 64 in order to prevent above case.

We have assigned 64000 to total transaction number in order to run at least a thousand transactions per client. Small number of transaction per client makes start up/shutdown transients overwhelm the steady state of data.

Set times is the number that states how many times to repeat the same test. We have stated this repeat rate as seven in order to get consistent TPS rates.

We have installed *pgbench* tool after the database installation completed and we have checked the TPS calculation from the source code. Standard *pgbench* code calculates the TPS rate by using wall clock time. We have introduced three variables which are *clock_t* type in order to calculate CPU Time and TPS rate calculated by using CPU Time. We have utilized *clock()* function which determines the amount of processor time used. We have crosschecked the calculated CPU Time with the one displayed with Linux *top* command and assured that the function calculates the CPU time. We have placed *clock()* function to starting and ending points of TPS calculation code block. Resulting time is divided by *CLOCKS_PER_SEC* in order to gather CPU Time in seconds.

Number of cores is another variable that we have evaluated during benchmarking. Taskset command is utilized to set the processor affinity for postgres processes.

Taskset is used to set or retrieve the CPU affinity of a running process given its PID or to launch a new COMMAND with a given CPU affinity. CPU affinity is a scheduler property that "bonds" a process to a given set of CPUs on the system. The Linux scheduler will honour the given CPU affinity and the process will not run on any other CPUs.

The CPU affinity is represented as a bit mask, with the lowest order bit corresponding to the first logical CPU and the highest order bit corresponding to the last logical CPU. When

taskset returns, it is guaranteed that the given program has been scheduled to a legal CPU [14].

2.7 Compute Resources vs Database Engine Performance

In this section benchmarking test results will be introduced.

PostgreSQL performance is first evaluated with the default settings as it is installed, after that memory parameters are upgraded and benchmarking tests are repeated. Finally, performance tests are evaluated by changing the number of cores assigned to postgres processes on memory-upgraded version.

2.7.1 Standard Version Details and Results

Standard version is the *PostgreSQL*-8.3.9 version that comes with default memory parameters. Three memory parameters affect the performance of *PostgreSQL* database. These are shared_buffers, work_mem and effective_cache_size. The standard version test has run on 64-core with default memory settings.

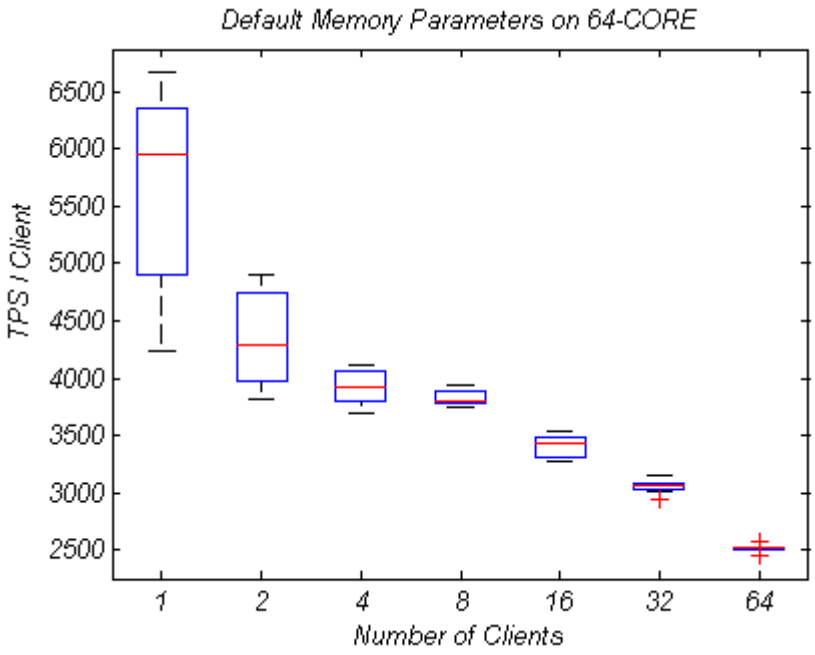


Figure 2.7 : Default Memory Parameters on 64-CORE.

2.7.1.1 shared_buffers (integer)

Sets the amount of memory the database server uses for shared memory buffers. The default is typically 32 megabytes (32MB), but might be less if your kernel settings will not support it (as determined during initdb). This setting must be at least 128 kilobytes. However, settings significantly higher than the minimum are usually needed for good performance. Several

tens of megabytes are recommended for production installations. This parameter can only be set at server start [2].

2.7.1.2 work_mem (integer)

Specifies the amount of memory to be used by internal sort operations and hash tables before switching to temporary disk files. The value defaults to one megabyte (1MB). Note that for a complex query, several sorts or hash operations might be running in parallel; each one will be allowed to use as much memory as this value specifies before it starts to put data into temporary files. Also, several running sessions could be doing such operations concurrently. So the total memory used could be many times the value of work_mem; it is necessary to keep this fact in mind when choosing the value. Sort operations are used for ORDER BY, DISTINCT, and merge joins. Hash tables are used in hash joins, hash-based aggregation, and hash-based processing of IN sub queries [2].

2.7.1.3 effective_cache_size (integer)

Sets the planner's assumption about the effective size of the disk cache that is available to a single query. This is factored into estimates of the cost of using an index; a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. When setting this parameter you should consider both *PostgreSQL*'s shared buffers and the portion of the kernel's disk cache that will be used for *PostgreSQL* data files. Also, take into account the expected number of concurrent queries on different tables, since they will have to share the available space. This parameter has no effect on the size of shared memory allocated by *PostgreSQL*, nor does it reserve kernel disk cache; it is used only for estimation purposes. The default is 128 megabytes (128MB) [2].

2.7.2 Memory-Upgraded Version results

In memory-upgraded version memory settings are upgraded to below parameters and same benchmarking on 64-core has run. Results can be found below.

Memory-Upgrade Values		
shared_buffers	:	8192MB
work_mem	:	1024MB
effective_cache_size	:	32768MB

Figure 2.8 : Memory Upgrade Values.

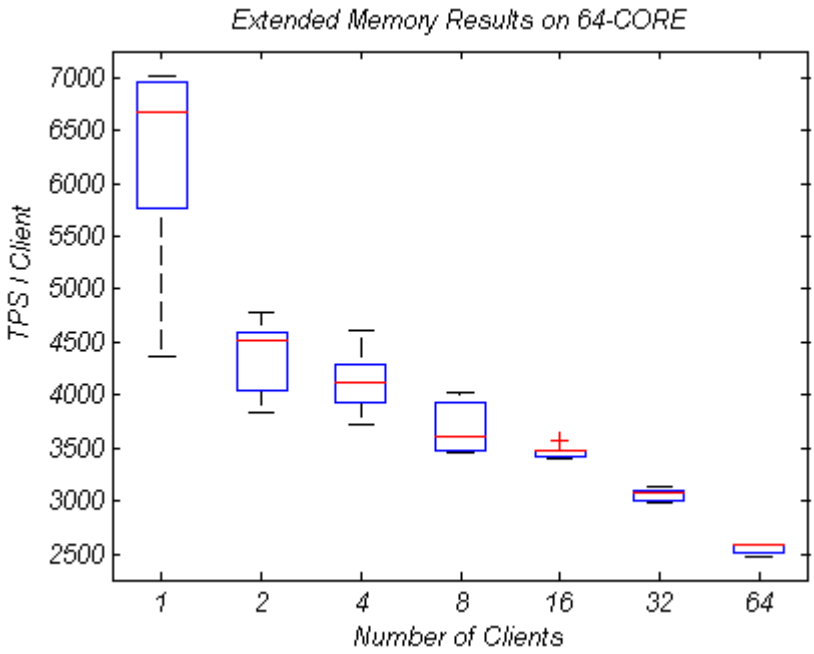


Figure 2.9 : Extended Memory Results on 64-CORE.

2.7.3 Results for Different Core Numbers

Benchmarking tests are run on the memory-upgraded version for different core numbers.

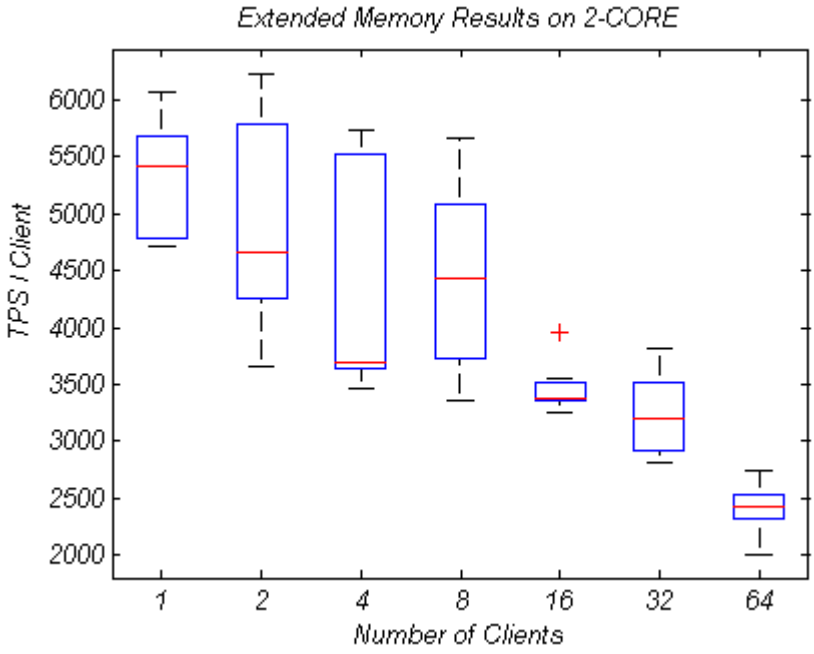


Figure 2.10 : Extended Memory Results on 2-CORE.

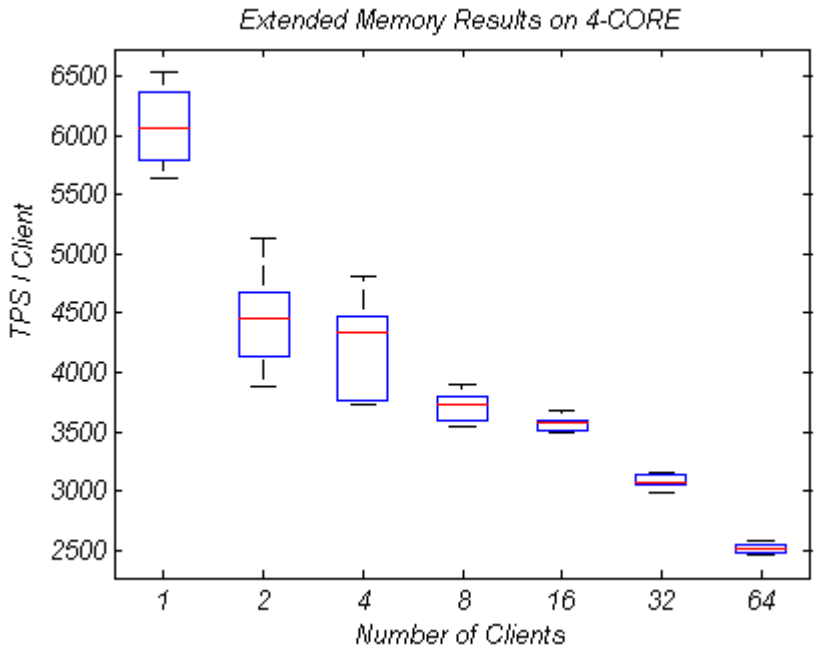


Figure 2.11 : Extended Memory Results on 4-CORE.

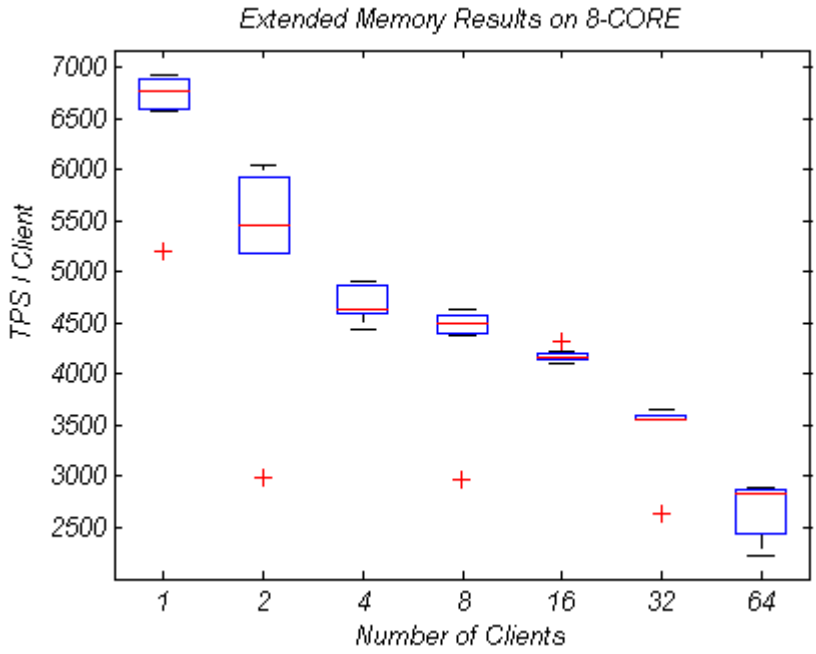


Figure 2.12 : Extended Memory Results on 8-CORE.

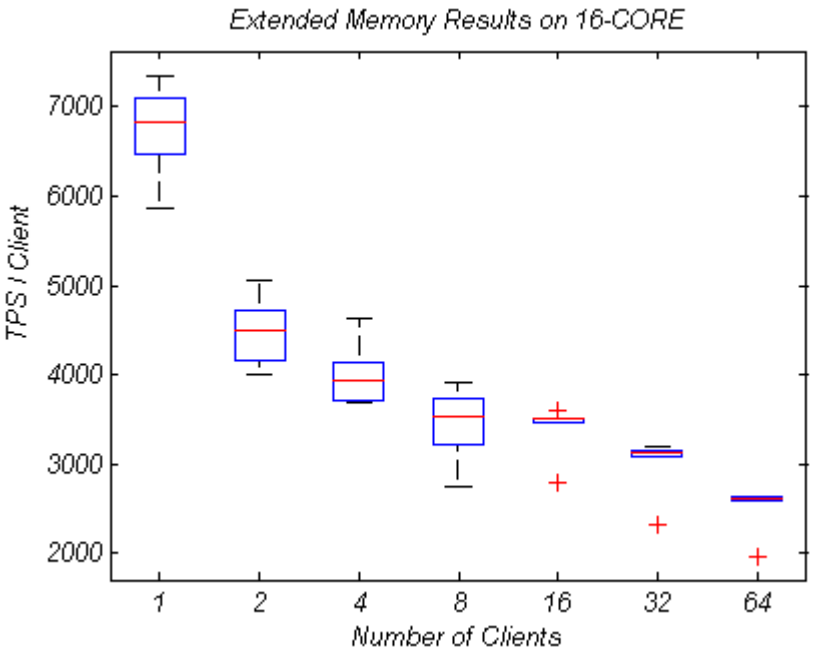


Figure 2.13 : Extended Memory Results on 16-CORE.

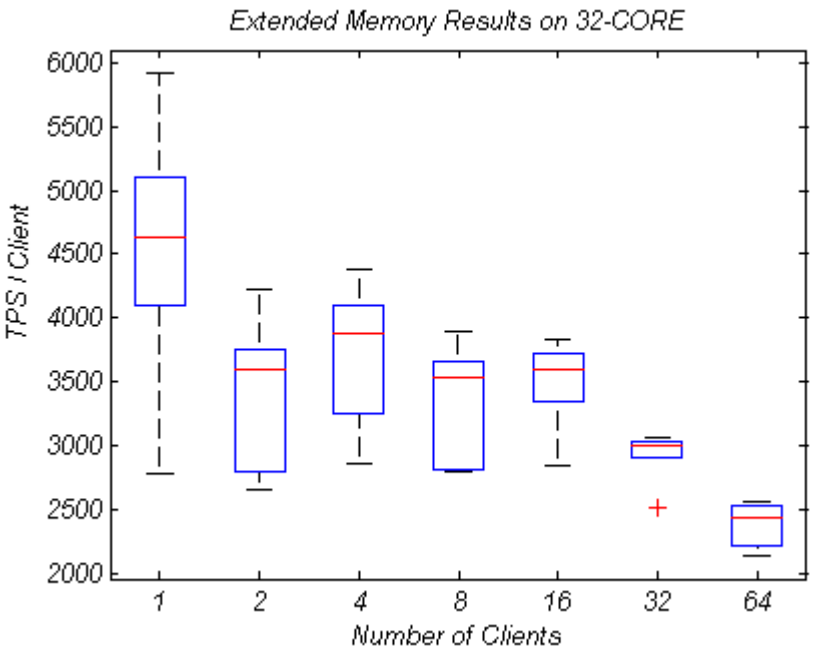


Figure 2.14 : Extended Memory Results on 32-CORE.

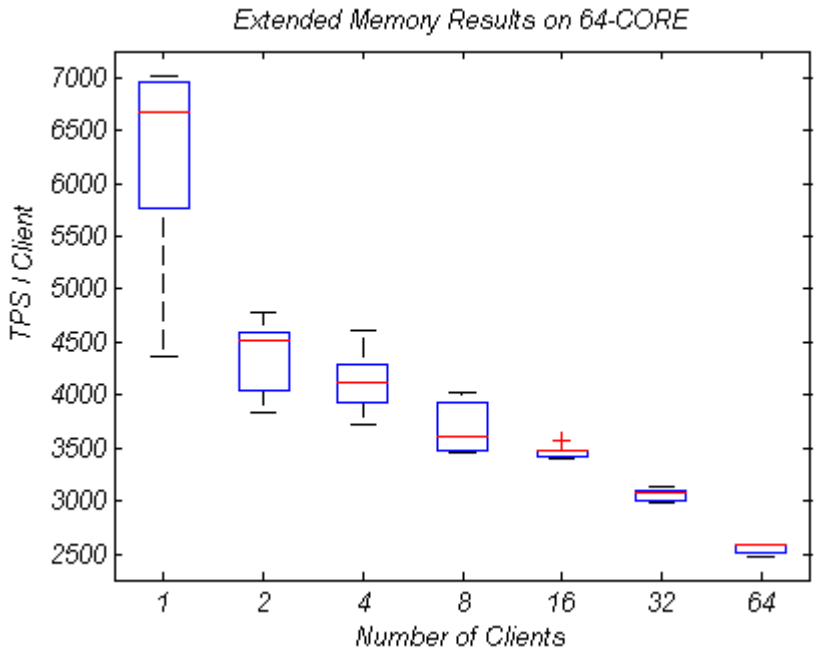


Figure 2.15 : Extended Memory Results on 64-CORE.

3. PARALELLIZING POSTGRESQL DATABASE ENGINE

3.1 Objectives

The focus of this chapter is to present the parallelization work and results accordingly.

3.2 Methodology

We have used *OpenMP* to rewrite *PostgreSQL*'s time-consuming functions. It is found that there is a difference between writing a code from scratch for parallelism and retrofitting it into an existing code [15]. The challenging part of the work done is to maintain the structure of a serial program when introducing parallelism. At that point, *OpenMP* excels, when compared to writing a hand-threaded program such as Pthreads.

3.2.1 Summary of *OpenMP* Pragma Directives

OpenMP (Open Multi-Processing) is an application-programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and FORTRAN on many architectures, including Unix and *Microsoft* Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behaviour. *OpenMP* is an implementation of multithreading, a method of parallelization whereby the master "thread" (a series of instructions executed consecutively) "forks" a specified number of slave "threads" and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors. In Figure 3.1 an illustration of multithreading can be can be found where the master thread forks off a number of threads [16].

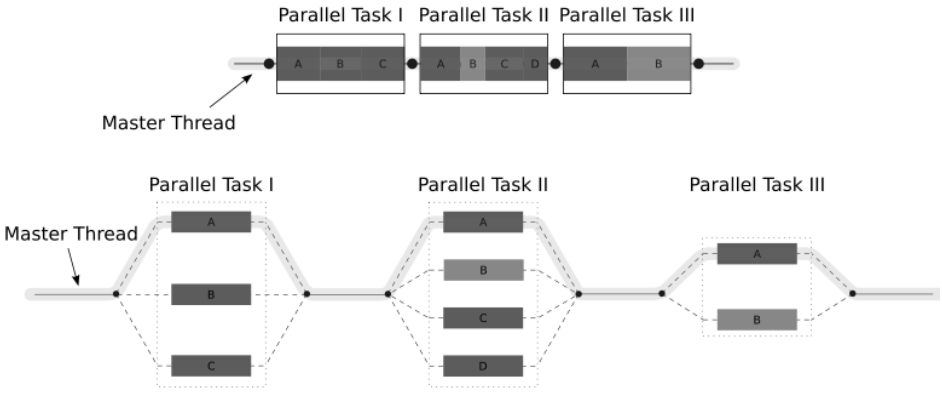


Figure 3.1 : An illustration of Multithreading.

The *OpenMP* specification defines a set of pragmas. A pragma is compiler directives on how to process the block of code that follows the pragma. The most basic pragma is the #pragma omp parallel to denote a parallel region. *OpenMP* supports two basic kinds of work-sharing constructs to specify that work in a parallel region is to be divided among the threads in the team. These work-sharing constructs are loops and sections. The #pragma omp for is used for loops, and #pragma omp sections is used for sections -- blocks of code that can be executed in parallel.

The #pragma omp barrier instructs all threads in the team to wait for each other before they continue execution beyond the barrier. There is an implicit barrier at the end of a parallel region. The #pragma omp master instructs the compiler that the following block of code is to be executed by the master thread only. The #pragma omp single indicates that only one thread in the team should execute the following block of code; this thread may not necessarily be the master thread. You can use the #pragma omp critical pragma to protect a block of code that should be executed by a single thread at a time [17].

3.2.2 How *PostgreSQL* Processes a Query

In this section, we will analyze the query execution of *PostgreSQL*. Below is the query processing diagram of *PostgreSQL*.

A query comes to the backend via data packets arriving through TCP/IP or Unix Domain sockets. It is loaded into a string, and passed to the parser, where the lexical scanner, scan.l, breaks the query up into tokens (words). The parser uses gram.y and the tokens to identify the query type, and load the proper query-specific structure, like CreateStmt or SelectStmt.

The statement is then identified as complex (SELECT / INSERT / UPDATE / DELETE) or a simple, e.g. CREATE USER, ANALYZE, etc. Simple utility commands are processed by statement-specific functions in backend/commands. Complex statements require more handling.

The parser takes a complex query, and creates a Query structure that contains all the elements used by complex queries. Query.qual holds the WHERE clause qualification, which is filled in by transformWhereClause(). Each table referenced in the query is represented by a RangeTableEntry, and they are linked together to form the range table of the query, which is generated by transformFromClause(). Query.rtable holds the query's range table.

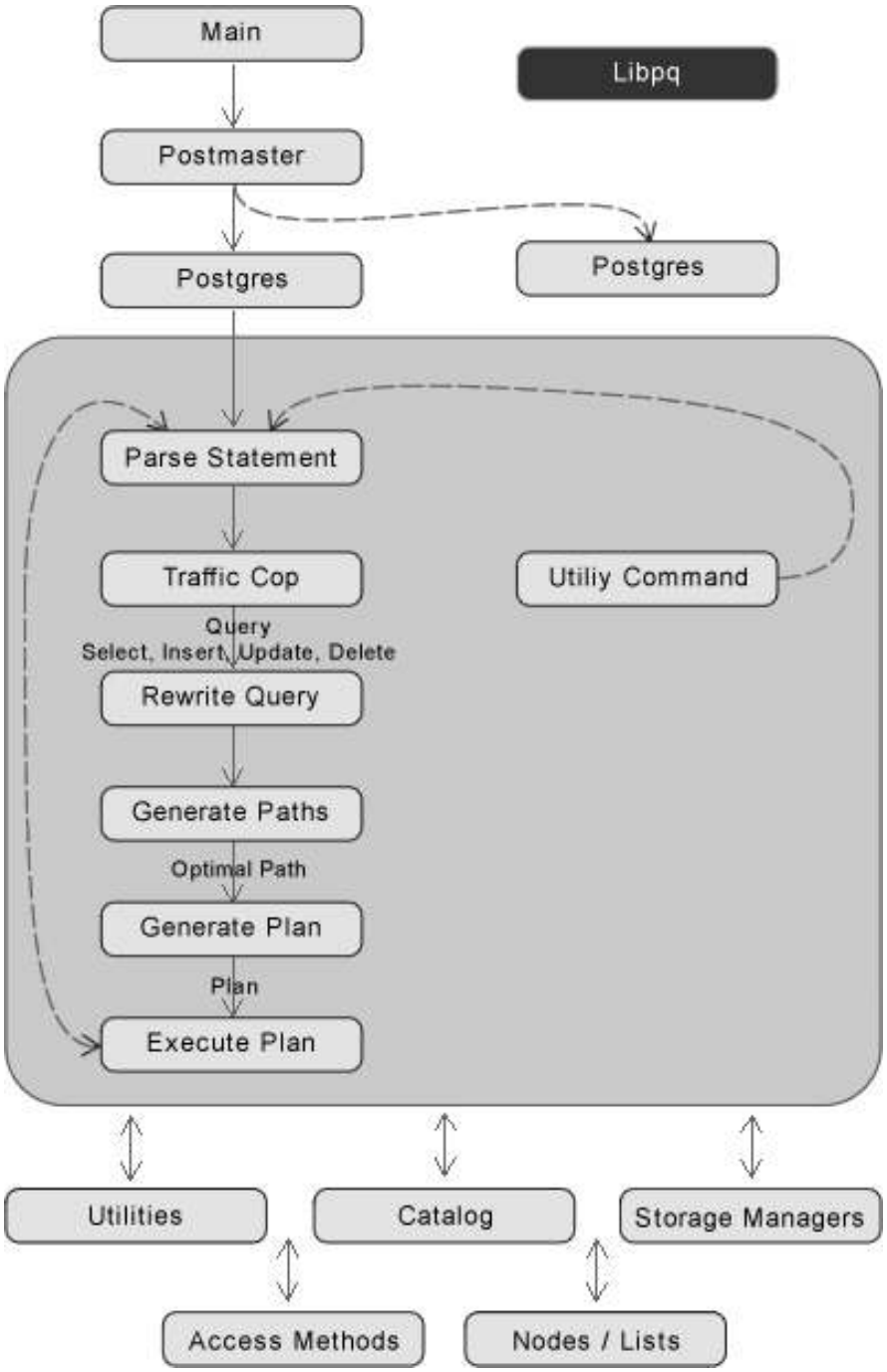


Figure 3.2 : *PostgreSQL* Query Processing.

Certain queries, like SELECT, return columns of data. Other queries, like INSERT and UPDATE, specify the columns modified by the query. These column references are converted to TargetEntry entries, which are linked together to make up the target list of the query. The target list is stored in Query.targetList, which is generated by transformTargetList().

Other query elements, like aggregates SUM(), GROUP BY, and ORDER BY are also stored in their own Query fields.

The next step is for the Query to be modified by any VIEWS or RULES that may apply to the query. This is performed by the rewrite system.

The optimizer takes the Query structure and generates an optimal Plan, which contains the operations to be performed to execute the query. The path module determines the best table join order and join type of each table in the RangeTable, using Query.qual (WHERE clause) to consider optimal index usage.

The Plan is then passed to the executor for execution, and the result returned to the client. The Plan is actually as set of nodes, arranged in a tree structure with a top-level node, and various sub-nodes as children [2].

3.3 Parallelization

We have started parallelization with time-consuming functions of Select and Insert operations. First we have tried to parallelize **slot_deform_tuple** function's for loop. It loops over the tuple and processing for each field depends completely on the previous one. Chained dependency in slot_deform_tuple has prevented the parallelization. In order to succeed in parallelizing *PostgreSQL* source code, we need loops where each iteration of the loop can be processed independently. Such loops can be found in **executor** code block. Executor is responsible of executing the query plan and producing any resulting tuples. In our study we have parallelized three for loops in executor. ExecMain.c and ExecQual.c source code files comprise the parallelized functions. We have verified the query results every time we parallelize a loop. A control query set consisting of create, insert and select functions has been constructed to check the accuracy of database operations.

As an addition to three for loops, we have also parallelized another four functions within executor code block. These are ExecMakeFunctionResultNoSets, ExecSelect, ExecScan and ExecStoreTuple. ExecMakeFunctionResultNoSets function is the most time-consuming function for Select operation and also it is the fourth time consuming one for Insert operation. Therefore we have decided to focus on that function and defined a parallel sections region for two independent code lines within the stated function. We have also analyzed ExecStoreTuple and ExecScan functions which are listed in the profiling results as time-consuming. These two functions are also parallelized with omp sections work-sharing construct.

In means of parallelization, we have three different versions. First, we have parallelized for loops with shared and private options and parallel sections are created without nowait option.

We have observed some performance problems with that version, tests were taking too much time to be completed. This is the first version that we have constructed.

Then we have created a second version in which we have improved *OpenMP* code blocks in terms of performance.

First, we have used nowait construct with parallel sections which eliminates redundant and unnecessary barriers.

Secondly, we have used firstprivate variable instead of shared in for loops. We have analyzed shared variables that are defined within for loops. If a shared variable in a parallel region is read by the threads executing the region, but not written to by any of the threads, then we have specified that variable to be firstprivate instead of shared. This avoids accessing the variable by dereferencing a pointer, and avoids cache conflicts. And lastly we have minimized the use of critical construct which slows down the parallel execution of source code.

We have run our control queries after these improvements and crosscheck the accuracy of transactions.

And in last version, we have just included the parallelized for loops. We have constructed third version by excluding parallel sections. Below details of parallel and serial codes can be found.

Source File : execTuples.c
Function Name : ExecStoreTuple

Serial Code

```
if (slot->tts_shouldFreeMin)

heap_free_minimal_tuple(slot-
>tts_mintuple);

/*
 * Store the new tuple into the
specified slot.
 */

slot->tts_isempty = false;

slot->tts_shouldFree = shouldFree;

slot->tts_shouldFreeMin = false;

slot->tts_tuple = tuple;

slot->tts_mintuple = NULL;

/* Mark extracted state invalid */

slot->tts_nvalid = 0;
```

Parallelized Code

```
/*
 * Store the new tuple into the
specified slot.
 */
#pragma omp parallel
{
  #pragma omp sections nowait
  {
    #pragma omp section
      slot->tts_isempty = false;

    #pragma omp section
      slot->tts_shouldFree =
shouldFree;

    #pragma omp section
      slot->tts_shouldFreeMin = false;

    #pragma omp section
      slot->tts_tuple = tuple;

    #pragma omp section
      slot->tts_mintuple = NULL;

    #pragma omp section
      slot->tts_nvalid = 0;
  }}
```

Figure 3.3 : Parallelized version of function ExecStoreTuple.

Source File : execScan.c
Function Name : ExecScan

Serial Code

```
qual = node->ps.qual;
projInfo = node->ps.ps_ProjInfo;

/*
 * If we have neither a qual to check
nor a projection to do, just skip
 * all the overhead and return the raw
scan tuple.
 */

if (!qual && !projInfo)
   return (*accessMtd) (node);
/*
 * Check to see if we're still
projecting out tuples from a previous
scan
 * tuple (because there is a function-
returning-set in the projection
 * expressions).  If so, try to project
another one.
 */

if (node->ps.ps_TupFromTlist)
{
   Assert(projInfo);
   /*
    * can't get here if not projecting
    */

   resultSlot = ExecProject(projInfo,
&isDone);

   if (isDone == ExprMultipleResult)
      return resultSlot;
      /*
       * Done with that source tuple...
       */
   node->ps.ps_TupFromTlist = false;
}
```

Parallelized Code

```
/*
 * Fetch data from node
 */
#pragma omp parallel
{
   #pragma omp sections nowait
   {
      #pragma omp section
       qual = node->ps.qual;

      #pragma omp section
       rojInfo = node->ps.ps_ProjInfo;
   }
}
```

Figure 3.4 : Parallelized version of function ExecScan.

Source File : execQual.c
Function Name : ExecMakeFunctionResultNoSets

Serial Code

```
InitFunctionCallInfoData(fcinfo,
&(fcache->func), i, NULL, NULL);
/*
 * If function is strict, and there are
any NULL arguments, skip calling
 * the function and return NULL.
 */
if (fcache->func.fn_strict)
{
   while (--i >= 0)
   {
      if (fcinfo.argnull[i])
      {
         *isNull = true;
         return (Datum) 0;
      }
   }
}

/* fcinfo.isnull = false;
 * handled by InitFunctionCallInfoData
 */

result = FunctionCallInvoke(&fcinfo);

*isNull = fcinfo.isnull;
```

Parallelized Code

```
/* fcinfo.isnull = false;
 * handled by InitFunctionCallInfoData
 */
#pragma omp parallel
{
   #pragma omp sections nowait
   {
      #pragma omp section
       result =
FunctionCallInvoke(&fcinfo);

      #pragma omp section
       *isNull = fcinfo.isnull;
   }
}
```

Figure 3.5 : Parallelized version of function ExecMakeFunctionResultNoSets.

Source File : execQual.c
Function Name : ExecEvalConvertRowtype

Serial Code

```
heap_deform_tuple(&tmptup,
                  cstate->indesc,
                  invalues + 1,
                  inisnull + 1);

invalues[0] = (Datum) 0;

inisnull[0] = true;

/*
 * Transpose into proper fields of the
new tuple.
 */
for (i = 0; i < outnatts; i++)
{
   Int j = attrMap[i];

   outvalues[i] = invalues[j];

   outisnull[i] = inisnull[j];
}
```

Parallelized Code

```
/*
 * Transpose into proper fields of the
new tuple.
 */
#pragma omp parallel
firstprivate(outnatts) private(i,
attrMap, outvalues, invalues,
outisnull, inisnull)
{
   int j;

   #pragma omp for
   for (i = 0; i < outnatts; i++)
   {
      j = attrMap[i];
      outvalues[i] = invalues[j];
      outisnull[i] = inisnull[j];
   }
}
```

Figure 3.6 : Parallelized version of function ExecEvalConvertRowtype.

Source File : execQual.c
Function Name : ExecEvalArray

Serial Code	Parallelized Code

`for (i = 0; i < outer_nelems; i++)`
`{`
`  memcpy(dat, subdata[i],`
`subbytes[i]);`
 	
`  dat += subbytes[i];`
 	
`  if (havenulls)`
`  array_bitmap_copy(`
`  ARR_NULLBITMAP(result),`
`  iitem,`
`  subbitmaps[i],`
`  0,`
`  subnitems[i]);`
 	
`  iitem += subnitems[i];`
`}`
	`#pragma omp parallel`
	`firstprivate(outer_nelems, havenulls)`
	`private(i, dat, subdata, subbytes,`
	`iitem, subbitmaps, subnitems)`
	`{`
	`  #pragma omp for`
	`  for (i = 0; i < outer_nelems; i++)`
	`  {`
	`  memcpy(dat,`
	`  subdata[i],`
	`  subbytes[i]);`
	
	`  dat += subbytes[i];`
	
	`  if (havenulls)`
	`  array_bitmap_copy(`
	`  ARR_NULLBITMAP(result),`
	`  iitem,`
	`  subbitmaps[i],`
	`  0,`
	`  subnitems[i]);`
	
	`  iitem += subnitems[i];`
	`  }`
	`}`

Figure 3.7 : Parallelized version of function ExecEvalArray.

Source File : execMain.c
Function Name : ExecSelect

Serial Code	Parallelized Code

`ExecSelect(TupleTableSlot *slot,`
`  DestReceiver *dest,`
`  EState *estate)`
`{`
`  (*dest->receiveSlot) (slot, dest);`
 	
`  IncrRetrieved();`
 	
`  (estate->es_processed)++;`
`}`
	`#pragma omp parallel`
	`{`
	`  #pragma omp sections nowait`
	`  {`
	`  #pragma omp section`
	`  (*dest->receiveSlot) (slot,`
	`dest);`
	
	`  #pragma omp section`
	`  IncrRetrieved();`
	
	`  #pragma omp section`
	`  (estate->es_processed)++;`
	`  }`
	`}`

Figure 3.8 : Parallelized version of function ExecSelect.

Source File : execMain.c
Function Name : ExecRelCheck

Serial Code

```
/* And evaluate the constraints */
for (i = 0; i < ncheck; i++)
{
   qual = resultRelInfo ->
ri_ConstraintExprs[i];

   /*
    * NOTE: SQL92 specifies that a
NULL result from a constraint
    * expression is not to be treated
as a
failure.  Therefore, tell
    * ExecQual to return TRUE for
NULL.
    */

   if (!ExecQual(qual, econtext,
true))
      return check[i].ccname;
}
```

Parallelized Code

```
#pragma omp parallel
firstprivate(ncheck) private(i, qual,
check, estate)
{
   #pragma omp for
   for (i = 0; i < ncheck; i++)
   {
      /*
       * ExecQual wants implicit-AND
form
       */
      qual = make_ands_implicit
(stringToNode(check[i].ccbin));

      resultRelInfo-
>ri_ConstraintExprs[i] = (List *)

      ExecPrepareExpr((Expr *) qual,
estate);
   }
}
```

Figure 3.9 : Parallelized version of function ExecRelCheck.

3.4 Parallel Results

In results section, we have only included second and third versions' of results. *OpenMP* Version 2 is the optimized state of Version 1 therefore; we have just included the Version 2.

Table 3.1: Results in terms of CPU Time –Standard Version.

Core	Client						
	1	2	4	8	16	32	64
8	9.8594	12.8649	13.6357	15.2171	15.3043	18.7523	24.3334
16	9.5404	14.3530	16.1013	18.6907	18.8713	21.4709	25.5969
32	14.8027	19.3657	17.5230	19.7350	18.4813	22.0077	26.9266
64	10.5253	14.6989	15.5334	17.3784	18.4739	20.8593	25.0291

Table 3.2: Results in terms of CPU Time –*OpenMP* Version 2.

Core	Client						
	1	2	4	8	16	32	64
8	15.9260	21.2040	25.1670	23.4710	25.6700	29.5280	37.1550
16	10.0120	18.8620	21.3930	22.4660	24.7400	28.5590	34.7500
32	10.1850	20.9840	23.9310	24.7680	26.4780	29.1730	35.2100
64	8.8892	13.0169	12.4890	15.0310	19.3300	25.1826	38.4984

Table 3.3: Results in terms of CPU Time –*OpenMP* Version 3.

Core	Client						
	1	2	4	8	16	32	64
8	15.4499	23.2130	19.4130	22.8359	22.5350	27.7979	33.2400
16	23.0629	23.6980	23.7729	22.9089	23.1389	27.1460	33.5240
32	8.5890	15.8889	23.2920	22.6110	22.7180	21.2669	27.2500
64	16.1370	16.6810	20.4059	20.5820	19.9579	22.1280	25.9909

Table 3.4: Comparison of Standard Version and *OpenMP* Version 2.

Core	Client						
	1	2	4	8	16	32	64
8	61.53%	64.82%	84.57%	54.24%	67.73%	57.46%	52.69%
16	4.94%	31.42%	32.87%	20.20%	31.10%	33.01%	35.76%
32	-31.20%	8.36%	36.57%	25.50%	43.27%	32.56%	30.76%
64	-15.54%	-11.44%	-19.60%	-13.51%	4.63%	20.73%	53.81%

Table 3.5: Comparison of *OpenMP* Version 2 and *OpenMP* Version 3.

Core	Client						
	1	2	4	8	16	32	64
8	-2.99%	9.47%	-22.86%	-2.71%	-12.21%	-5.86%	-10.54%
16	130.35%	25.64%	11.12%	1.97%	-6.47%	-4.95%	-3.53%
32	-15.67%	-24.28%	-2.67%	-8.71%	-14.20%	-27.10%	-22.61%
64	81.53%	28.15%	63.39%	36.93%	3.25%	-12.13%	-32.49%

Comparison tables indicate that *OpenMP* Version 3 increases the performance for more points compared to Version 2. For Version 2 on 64-Core, percentage changes show that on average there is a 14% decrease in CPU time up to 16 clients. In Version 3, we have excluded 'omp parallel sections' which can add overhead to the overall performance. Calling a task within a section just creates extra overhead and cannot control and synchronize the tasks since each parallel section is independent of each other. With parallel sections, there is no way to coordinate the task in each section, so it is not possible to determine whether one section will be executed before another, regardless of which section comes first in the program source. Tasking has much better performance and scalability for nested parallel and recursive algorithms, compared to parallel sections, but it is available at *OpenMP* 3.0. However, Standard Version is still performing better for most points when compared to Parallel Versions. Below TPS results are also calculated from the CPU Time generated therefore they are parallel to the above results. The consistency of the *OpenMP* Version 2 up to 16 clients can be observed by comparing the boxplot representation of Standard and OMP Version 2.

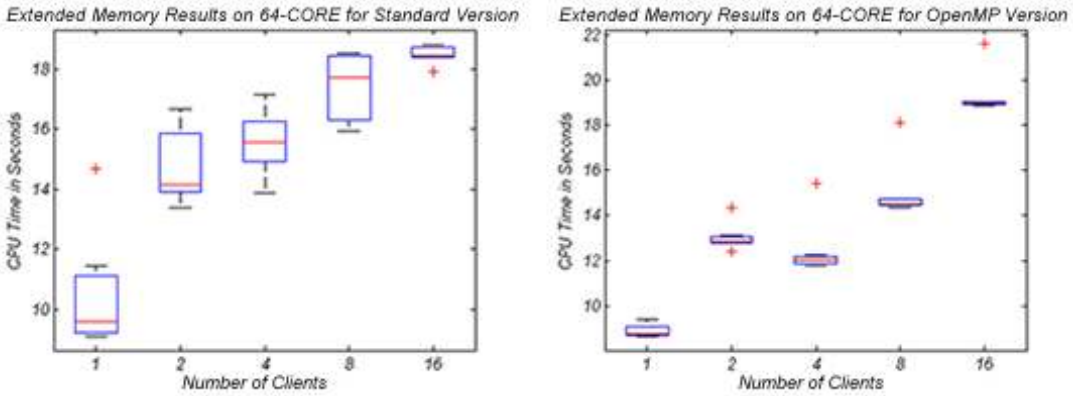


Figure 3.10 : Comparison of standard version and OMP version 2.

Table 3.6: Results in terms of TPS – Standard Version.

Core	Client						
	1	2	4	8	16	32	64
8	6549	5240	4698	4299	4182	3452	2654
16	6741	4484	3997	3468	3413	3014	2524
32	4544	3397	3732	3304	3493	2919	2388
64	6236	4380	4137	3696	3465	3069	2557

Table 3.7: Results in terms of TPS – *OpenMP* Version 2.

Core	Client						
	1	2	4	8	16	32	64
8	4018	3018	2542	2726	2493	2167	1722
16	5122	2889	2729	2733	2498	2216	1771
32	6283	3049	2674	2583	2417	2193	1817
64	7205	4925	5165	4283	3317	2541	1662

Table 3.8: Results in terms of TPS – *OpenMP* Version 3.

Core	Client						
	1	2	4	8	16	32	64
8	4142	2757	3296	2802	2840	2302	1925
16	2775	2700	2692	2793	2765	2357	1909
32	7451	4027	2747	2830	2817	3009	2348
64	3966	3836	3136	3109	3206	2892	2462

Table 3.9: Comparison of Standard Version and *OpenMP* Version 2.

Core	Client						
	1	2	4	8	16	32	64
8	-38.65%	-42.40%	-45.89%	-36.59%	-40.39%	-37.22%	-35.11%
16	-24.02%	-35.57%	-31.73%	-21.20%	-26.82%	-26.48%	-29.84%
32	38.26%	-10.24%	-28.35%	-21.83%	-30.81%	-24.87%	-23.90%
64	15.54%	12.45%	24.86%	15.89%	-4.26%	-17.18%	-35.01%

Table 3.10: Comparison of *OpenMP* Version 2 and *OpenMP* Version 3.

Core	Client						
	1	2	4	8	16	32	64
8	3.09%	-8.65%	29.66%	2.79%	13.92%	6.23%	11.79%
16	-45.82%	-6.54%	-1.36%	2.20%	10.69%	6.36%	7.79%
32	18.59%	32.08%	2.73%	9.56%	16.55%	37.21%	29.22%
64	-44.95%	-22.11%	-39.28%	-27.41%	-3.35%	13.81%	48.13%

We have decided to increase the database size and total transaction number in order to see the effect clearer for fewer sources like number of Cores 4 and 8. In order to increase the database size we have taken the scale as 640, which creates a database including 64.000.000 records and total transaction number is increased to 640.000. As you can see on Figure 3.11 and Figure 3.12 *OpenMP* Version 3 is the best performer.

This results show that *PostgreSQL* can get over the parallelization overhead when the table size and total transaction size are increased.

Table 3.11: Results in terms of CPU Time –Standard Version – Scale 640.

Core	Client						
	1	2	4	8	16	32	64
4	191.2119	239.4550	228.3930	226.4439	242.9950	302.5620	360.7819
8	162.9130	226.5629	217.7050	222.3549	235.1220	274.0070	327.2130

Table 3.12: Results in terms of CPU Time –*OpenMP* Version 2 – Scale 640.

Core	Client						
	1	2	4	8	16	32	64
8	232.6409	229.8199	235.8729	241.7690	250.9000	289.3079	352.2880

Table 3.13: Results in terms of CPU Time –*OpenMP* Version 3 – Scale 640.

Core	Client						
	1	2	4	8	16	32	64
4	199.2959	235.3820	226.7429	221.8470	245.6990	293.4130	363.3249
8	187.2520	209.3720	211.9579	213.0149	227.5200	261.9139	323.9610

Table 3.14: Results in terms of TPS –Standard Version – Scale 640.

Core	Client						
	1	2	4	8	16	32	64
4	3347	2672	2802	2826	2633	2115	1773
8	3928	2824	2939	2878	2721	2335	1955

Table 3.15: Results in terms of TPS Time –*OpenMP* Version 2 – Scale 640.

Core	Client						
	1	2	4	8	16	32	64
8	2751	2784	2713	2647	2550	2212	1816

Table 3.16: Results in terms of TPS Time –*OpenMP* Version 3 – Scale 640.

Core	Client						
	1	2	4	8	16	32	64
4	3211	2718	2822	2884	2604	2181	1761
8	3417	3056	3019	3004	2812	2443	1975

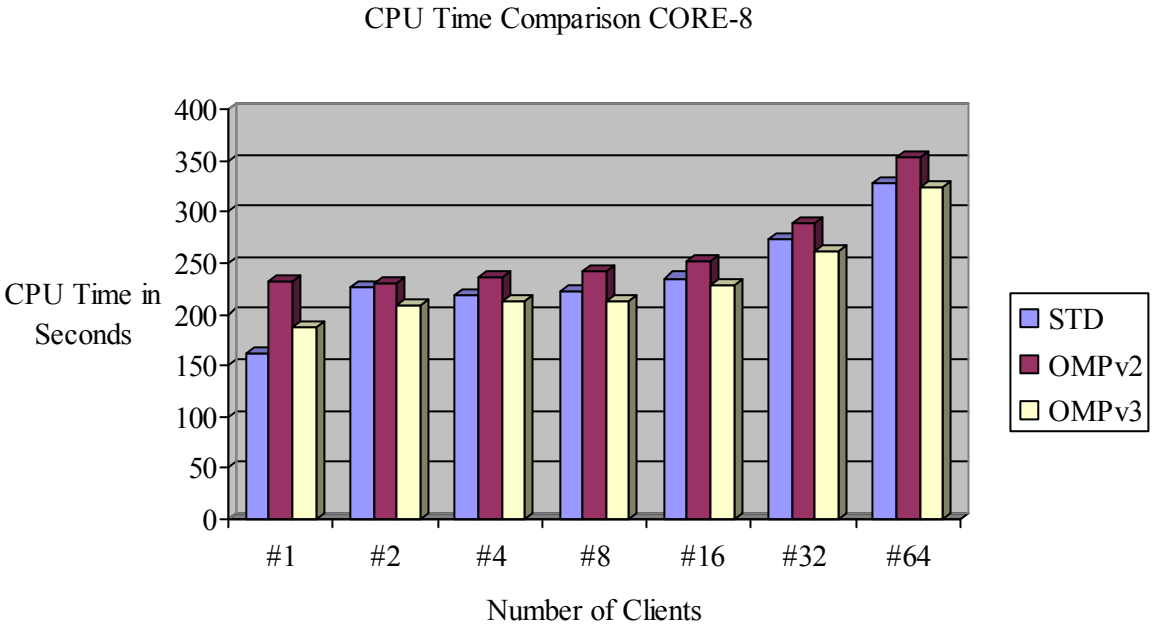


Figure 3.11 : Bar Chart of CPU Time Comparison for scale 640.

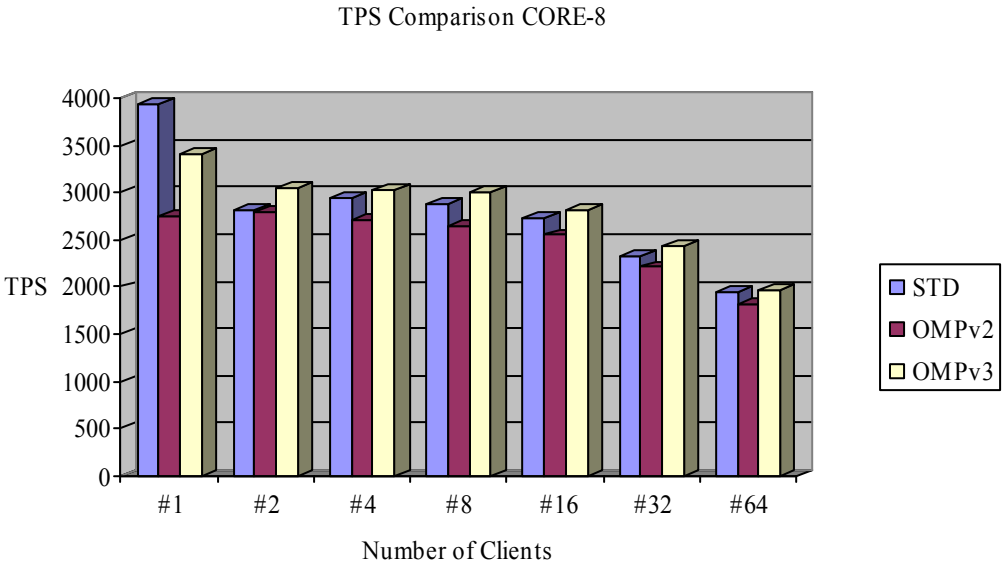


Figure 3.12 : Bar Chart of TPS Comparison for scale 640.

4. CONCLUSION AND RECOMMENDATIONS

4.1 Application of The Work

In this thesis, the necessary steps for parallelizing time-consuming functions of *PostgreSQL* database were discussed. We analyzed database performance with different tools like profilers and benchmarking tests. In doing so, our method consisted of the following steps:

- Preparing query sets in order to measure performance.
- Profiling *PostgreSQL* with *Gprof*
- Profiling *PostgreSQL* with *OProfile*
- Comparing profiling results and determining time consuming functions of source code
- Running benchmark tests with *pgbench*
- Recoding time consuming *PostgreSQL* functions
- Measuring performance in terms of TPS and CPU Time and comparing results.

4.2 Limitations

The source code that we have worked on has not been designed for parallelization. Therefore, this inherits problems, which make parallelization hard to implement.

Loops that are going to be parallelized should have independent iterations. Because of data dependency, some time-consuming functions are untouchable in terms of parallelization.

4.3 Conclusions

We have used various tools and methods to measure and improve the database performance. We have profiled the serial database performance and obtain the knowledge of time-consuming functions. We have applied a way of parallelization for suitable functions and when we see some performance issues on parallelized version, we optimized the *OpenMP* code blocks. A cost is associated with the creation of *OpenMP* parallel regions. The sources of overheads include the cost of starting up threads and creating the execution environment, the potential additional expense incurred by the encapsulation of a parallel region in a

separate function, the cost of computing the schedule, the time taken to block and unblock threads, and time for them to fetch work and signal that they are ready. We have observed that *PostgreSQL* get over that parallelization costs when the tuple number is increased. In addition, we have seen the positive effect of *OpenMP* optimization by using less critical regions, processing firstprivate instead of shared and adding nowait to parallel sections.

In summary, we have presented evidence of a positive effect of *OpenMP* implementation on *PostgreSQL*.

Results show that it is possible to implement multi-processing and obtain improved results in terms of CPU Time and TPS. However, it is hard to implement parallelization to databases in the current way that they have been built. In the future, databases need to be designed in a way to take advantage of multi-core architectures.

REFERENCES

[1] **Dikenelli, O., Ünalır, M.O., Özerdim, A., Özkarahan, E. A.,** 1995: Load Balancing Approach for Parallel Database Machines, IEEE.

[2] ***PostgreSQL*****,** <*http://www.PostgreSQL.org*>, accessed at 07.05.2010.

[3] ***OpenMP***, <*http://www.OpenMP.org/*>, accessed at 07.05.2010.

[4] ***Gprof***, <*http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html*>, accessed at 07.05.2010.

[5] ***OProfile*****,** <*http://OProfile.sourceforge.net/*>, accessed at 07.05.2010.

[6] ***Pgbench*****,**<*http://developer.PostgreSQL.org/pgdocs/postgres/pgbench.html*>, accessed at 07.05.2010.

[7] **Parallel Computing: Background,** <*http://www.intel.com/pressroom/kits/upcrc/parallelcomputing_backgrounder.pdf*>, accessed at 07.05.2010.

[8] **Taniar, D., Leung, C.H.C., Rahayu, W., Goel, S., 2008:** High-Performance Parallel Database Processing and Grid Databases. New Jersey: Wiley.

[9] **Popular Database Management Systems Overview**, <http://dbconvert.com/overview.php>, accessed at 07.05.2010.

[10] **Benchmarking**, <http://en.wikipedia.org/wiki/Benchmarking>, accessed at 07.05.2010.

[11] **Benchmarking Computing**, <http://en.wikipedia.org/wiki/Benchmark_(computing)>, accessed at 07.05.2010.

[12] **Database Benchmarking**, <http://wiki.*Oracle*.com/page/Database+Benchmarking>, accessed at 07.05.2010.

[13] **TPC**, <http://www.tpc.org/>, accessed at 07.05.2010.

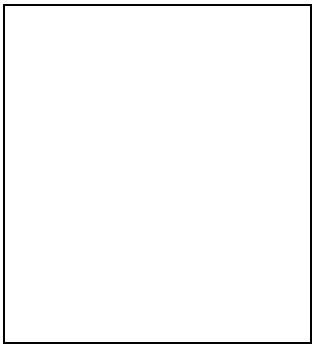
[14] **Taskset**, <www.linuxcommand.org>, accessed at 07.05.2010.

[15] **Bob Kuhn, Paul Petersen, Eamonn O'Toole,** 2000:: *OpenMP* versus threading in C/C++. Concurrency - Practice and Experience 12(12): 1165-1176 ()

[16] <http://en.wikipedia.org/wiki/*OpenMP*>, accessed at 07.05.2010.

[17] <www.developers.sun.com>, accessed at 07.05.2010.

CURRICULUM VITAE



Candidate's full name: Reydan Çankur

Place and date of birth: İstanbul / 15.08.1982

Permanent Address: Çınar Mah. Yeşiltepe Sk. No: 23 D: 6 Küçükyalı-MALTEPE/IST

Universities and
Colleges attended: Boğaziçi University – Management Information Systems (MIS)

Publication: Cankur, R., Dalfes, H.N. "Parallel Experiments on PostgreSQL." Poster presented as part of the PARA 2010: State of Art in Scientific and Parallel Computing, Reykjavik, Iceland, 6-9 June 2010.