A Parallel Concordance Benchmark

Haskell Implementation

Design

- **Data Structure :**

- Hash table is used for organizing the concordance data.
Two versions with different hash tables :
1- Haskell hash table **(Data.Hashtable)**
2- Glib Hash table through (FFI)
{-# LANGUAGE ForeignFunctionInterface #-}

Two main Phases :

1- ProduceSeq → generating list of pairs representing sequences with it start indices . [(String , Int)]
2- main_process → taking the list produced and insert or update the hash table . [(String , Int)] - > IO ()

- **<u>FFI</u>:**

-Haskell's FFI is used to call functions from other languages .

- foreign import ccall "ht_init" ht_init :: IO ()
- foreign import ccall "ht_insert" ht_insert :: CString -> Int -> IO ()

- Calling functions : ht_init and ht_insert from C code :
- void ht_init ()
- void ht_insert (char *key, int value)

- **<u>HardWare</u>** : The programs have been measured on a common multi-core architecture, eight-core machine, comprising two Intel Xeon E5410 quad-core processors, running at 2.33 GHz, with a 1998 MHz front-side bus 6144 KB and 8GB RAM running under CentOS release 5.5 .

- **<u>Software :</u>** The compiler used is the ghc version 6.12.3 . For the Glib based version , the Glib library version used is 2.24.1

- **<u>Text File :</u>** For the reported experiments , the text file used is : bible.txt (4.6 MB) which has 800000 words .

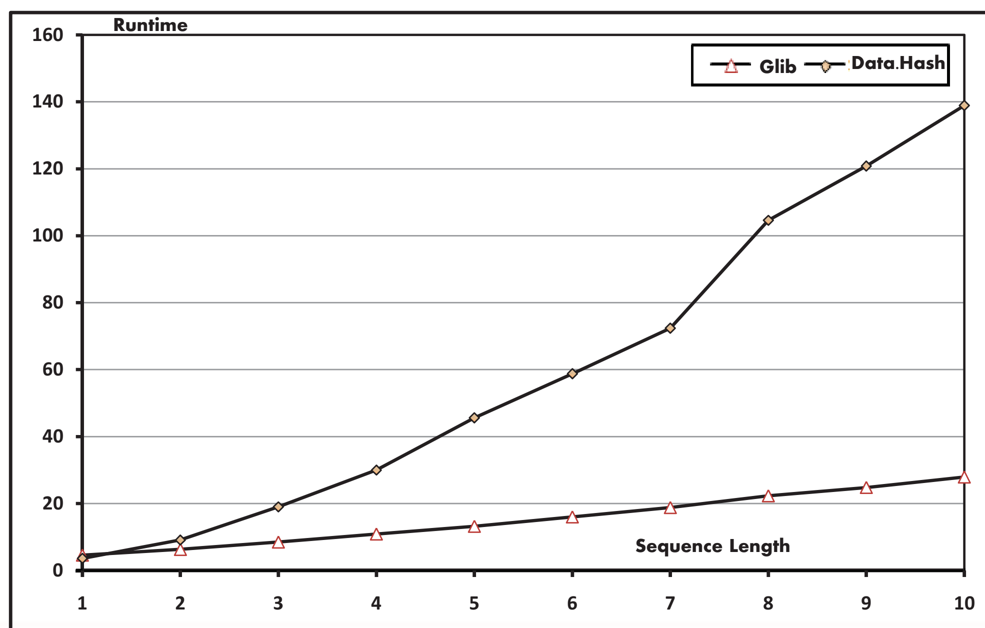
Sequential Implementation

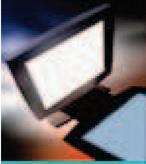
Data.Hashtable Vs Glib Hash table :

	Run Time	
N	Glib hash table	Data.Hashtable
1	4.6	3.6
2	6.3	9.1
3	8.5	19.0
4	10.9	30.0
5	13.2	45.6
6	16.0	58.8
7	18.8	72.4
8	22.3	104.6
9	24.8	120.8
10	27.9	138.9

Sequential Implementation

Data.Hashtable Vs Glib Hash table





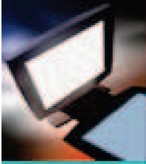
Parallel Implementation

Parallelizing ProduceSeq :

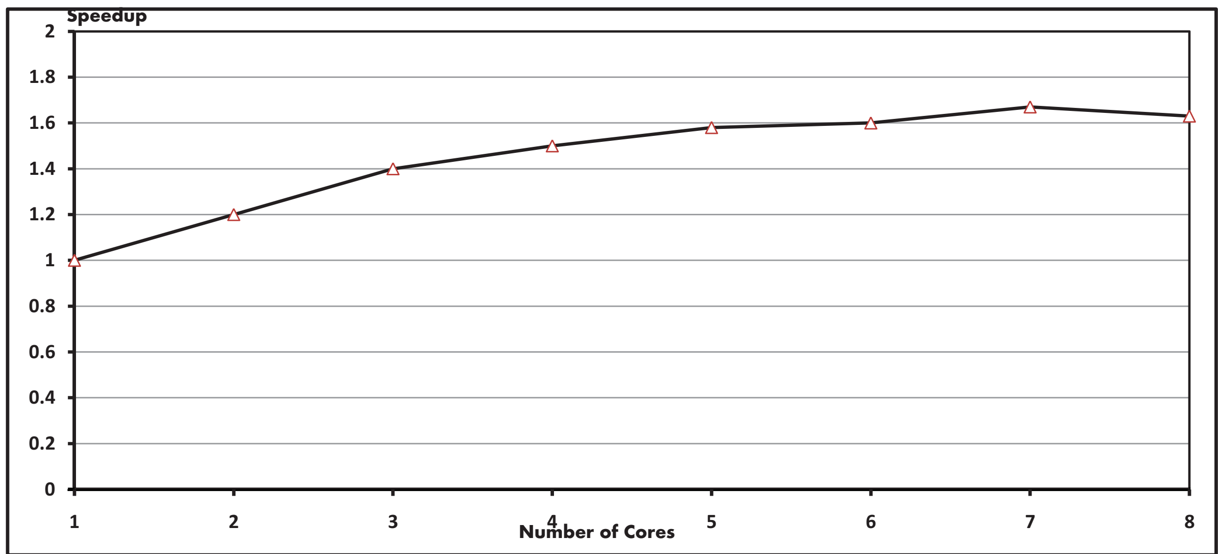
1- Glib based version :-

For sequence length = 7

	Glib –based	
N	RunTime	Relative Speedup
1	42.1	1
2	33.7	1.2
3	28.4	1.4
4	27.3	1.5
5	26.5	1.58
6	26.2	1.6
7	25.2	1.67
8	25.8	1.63



Parallel Implementation



Conclusion

- Performance difference between Data.Hashtable and using Glib through FFI .
- Using Lists for concordance is faster than both versions of hash tables Thomas code : http://www.mathematik.uni-marburg.de/~horstmey/sicsa/ConcordanceTH.hs
- Parallelizing (produceSeq function) of the Haskell code using strategies was simple to use and apply .

```
produceSeq n seq = (concat $ map tak tai) `using` parListChunk cSize rdeepseq
 where tak = (takesns n )
       tai = (tails seq)
       cSize = 1000
```

Conclusion

- **further parallelization** :

Parallelizing main_process function :
would need splitting hash table and Using locks
- Haskell-level locks
- C locks using mutex .

A Parallel Concordance Benchmark

OpenMP Implementation

Design

- **Data Structure :**
- Hash table is used for organizing the concordance data.
- Single linked list for storing the indices numbers since.

KEY	VALUE: Frequency	VALUE: Indices
A	4	→ [•] → [•] → ....
AN	2	→ [•] → [•]
BIG	13	→ [•] → [•] ....
FLOWER	3	→ [•] → [•] → [•]
MAN	23	→ [•] → [•] → ....
TALL	17	→ [•] → [•]
TOY	54	→ [•] → [•] → [•] → ....

Implementation

- OpenMP is a de facto standard (API) used mainly with shared memory architecture to provide parallel applications.
- It is a specification that can be added to some programming languages such as Fortran , C and C++ to specify the coordination aspects of a parallel program.
- OpenMP consists of a set of compiler directives, supporting library routines and environment variables to specify the parallelism, and program runtime characteristics .
- OpenMP is widely used for fine grain loop-level parallelism since it supports incremental development as well as being easy to implement .

  #pragma omp parallel for
- Some code changes beyond pragmas are needed for performance tuning .

Implementation

. The Hash table is implemented in C + OpenMP version using Glib hash table

- OpenMP is implemented in Single Program Multiple Data (SPMD) model by spawning the specified number of threads in the parallel region.
- Each thread uses its id value, for specifying the area of a text on which the thread has to work.
- This is based on the OpenMP parallel directive to encloses the parallel region.
- **#pragma omp parallel shared(sequences) private(file, tId , worker type , offset)**

Evaluation Platform

- **HardWare** : The programs have been measured on a common multi-core architecture, eight-core machine, comprising two Intel Xeon 5410 quad-core processors, running at 2.33 GHz, with a 1998 MHz front-side bus 6144 KB and 8GB RAM running under CentOS release 5.5 .
- **Software :** The compiler used is the gcc version 4.1.2 . for profiling the benchmarks , ompP 0.7.1 profiling tool [1] is used .
- **Text Files :** For all the reported experiments, two samples of files were used with different sequence lengths :

1. 18 MB and the sequences of up to 10 words.
2. 1 MB and the sequences of up to 50 words.

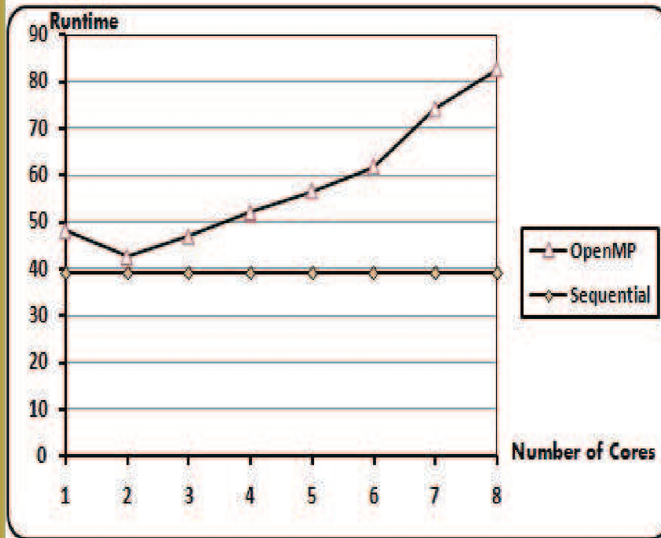
Profiling Tool (ompP)

- OmpP is a profiling tool designed to help the programmer to understand the scalability behavior of the OpenMP applications on shared memory architecture http://www.cs.utk.edu/~karl/ompp.html
- The ompP plays a great role in discovering and analyzing different kinds of overhead which limit the application's scalability.
- It determines the execution times for all OpenMP directives.
- It also analyses the overhead for each parallel region separately as well as for the whole program, and generates a profiling report upon the completion of the program execution
- The ompP overhead analysis report shows four overhead categories : Synchronization, Imbalance, Limited parallelism and Thread Management overhead

Implementation

Naïve Parallelism :

-One shared hash table protected using mutual exclusion construct :
#pragma omp critical
- Poor performance !



Number of Cores	Runtime (Seconds)
1	48.35
2	42.7
3	47
4	52.1
5	56.7
6	61.8
7	74.2
8	82.6
Sequential	39

Performance Analysis

- different Potential reasons for poor performance in OpenMP programs :

1. effect of the synchronization overhead -> waiting for long time
2. the critical construct is the most expensive synchronization construct supported by OpenMP
3. the compiler and the runtime system overhead

Performance Analysis

- ompP Profiling Results :

```
----------------------------------------------------------------------
----     ompP Overhead Analysis Report     ---------------------------
----------------------------------------------------------------------
Total runtime (wallclock)    : 271.48 sec [8 threads]
Number of parallel regions   : 2
Parallel coverage            : 263.58 sec (97.09%)

Parallel regions sorted by wallclock time:
          Type                          Location      Wallclock (%)
R00003  PARALLEL              firstomp.c (115-139)    263.39 (97.02)
R00001  PARALLEL              firstomp.c (101-103)      0.19 ( 0.07)
                                                 SUM   263.58 (97.09)

Overheads wrt. each individual parallel region:
          Total        Ovhds (%)  =   Synch  (%)  +  Imbal   (%)  +   Limpar (%)  +    Mgmt (%)
R00003  2107.11  1781.10 (84.53) 1388.33 (65.89)  256.31 (12.16)    0.00 ( 0.00)  136.45 ( 6.48)
R00001     1.56     1.56 (100.00)    0.00 ( 0.00)    0.47 (30.20)    0.00 ( 0.00)    1.09 (69.80)

Overheads wrt. whole program:
          Total        Ovhds (%)  =   Synch  (%)  +  Imbal   (%)  +   Limpar (%)  +    Mgmt (%)
R00003  2107.11  1781.10 (82.01) 1388.33 (63.92)  256.31 (11.80)    0.00 ( 0.00)  136.45 ( 6.28)
R00001     1.56     1.56 ( 0.07)    0.00 ( 0.00)    0.47 ( 0.02)    0.00 ( 0.00)    1.09 ( 0.05)
   SUM  2108.67  1782.66 (82.08) 1388.33 (63.92)  256.78 (11.82)    0.00 ( 0.00)  137.54 ( 6.33)
```


Performance Tuning

- One way of reducing the synchronization overhead is to divide the hash table into multiple hash tables, and use a different lock for each one .

```
i = select_hashtable (k)
#pragma omp critical (i)
        insert(sequences[i],k,current_s);
……
```

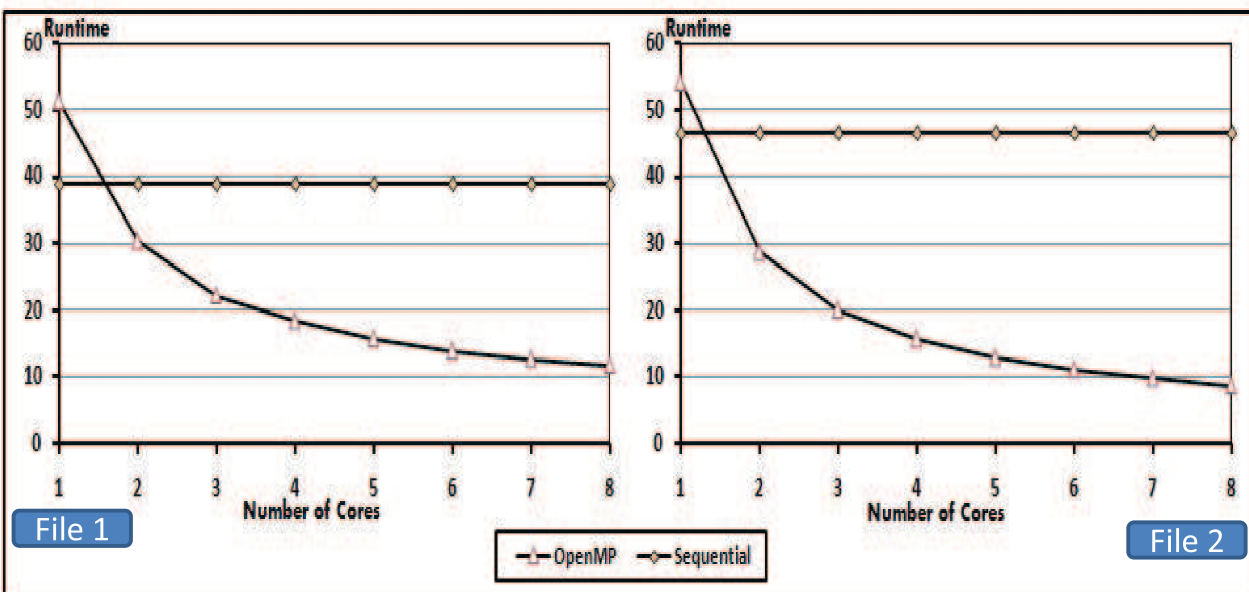
Performance Tuning

- -> significantly reducing the waiting time

Number of Hash tables	Synchronization Overhead	Speedup on 8 cores
1	63%	0.58
8	27.65%	1.65
54	21.6%	3.3
364	12.47%	4.2
657	4.86%	4.38

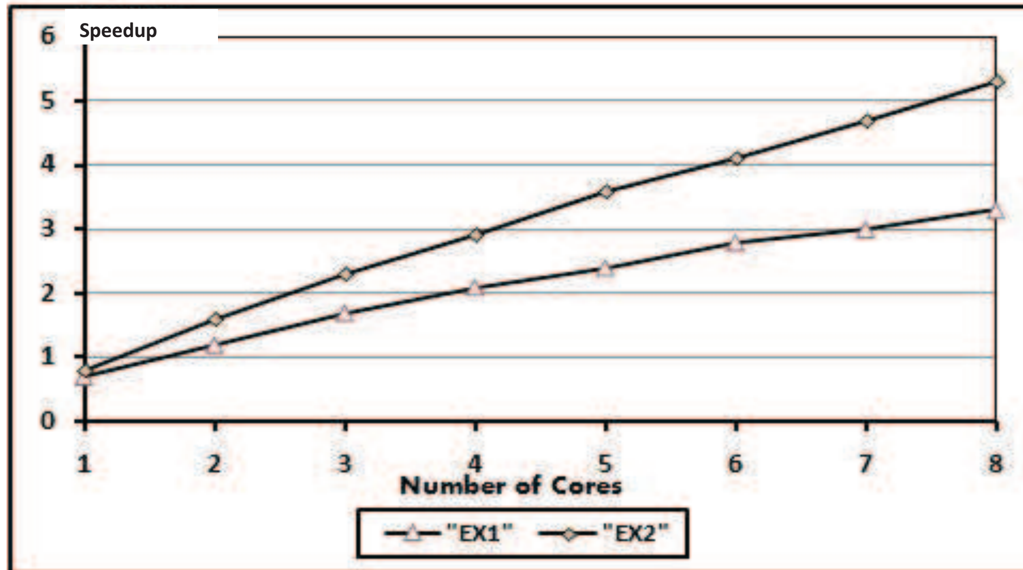
Final Results

- the measured runtime with the number of cores compared to the
- sequential version for the final OpenMP implementation for Experiment 1 and Experiment 2.



Final Results

- the speedup for the final OpenMP implementation of Experiment 1 and Experiment 2 .



Conclusion

- OpenMP is simple to learn and to use with little programming effort. Moreover, it provides high-performance applications that are able to be run on different shared memory platforms and by different numbers of threads.
- OpenMP allows parallelization to be carried out incrementally
- Although OpenMP is considered a high level parallel programming model, the parallelization task is not always easy and straightforward. The programmer still needs to think carefully of how to exploit parallelism efficiently , and reducing different kind of overheads .