2008 Ninth International Conference on Parallel and Distributed Computing, Applications and Technologies

# Efficient Parallelization of a Protein Sequence Comparison Algorithm on Manycore Architecture

Xiaochun Ye[1,2] Van Hoa Nguyen[3], Dominique Lavenier[3] Dongrui Fan[1]
*yexiaochun@ict.ac.cn* *{vhnguyen,lavenier}@irisa.fr* *fandr@ict.ac.cn*
*1. Key Lab of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China*
*2. Graduate University of Chinese Academy of Sciences, Beijing, China*
*3. IRISA / INRIA-CNRS, Rennes, France*

## Abstract

*This paper introduces the Godson-T manycore architecture and demonstrates the efficiency of its synchronization mechanism through a computation intensive bioinformatics application: the comparison of protein banks. The parallel part of the protein sequence comparison algorithm can nearly get a linear speed-up thanks to a fine tuning of the synchronization mechanism provided by the Godson-T chip.*

## 1. Introduction

In bioinformatics, one of the basic tasks is to compare genomic sequences. There are two main families of applications. The first one is devoted to the search of large databases. The second family concerns large bioinformatics treatments made on the newly data available every day. In that case, sequence comparison is often one of the first operations which are performed on raw data before more complex processing.

Biotechnology improvement in the sequencing area has led to a huge increase in the size of genomic databases such as GenBank and UniProt protein database. Being able to process these data as fast as possible, represent one of the current challenges in bioinformatics. More precisely, comparing sequence means extracting similar zone between sequences. The difficulty is that we do not know both the location and the size of these similar zones. Many algorithms have been proposed to solve this problem, including Smith-Waterman algorithm [1] and some other powerful heuristics ones such as Blast [2] and Fasta [3].

The algorithm we used in this paper belongs to the seed heuristics algorithm family. It has been developed at IRISA and specifically designed for various parallel platforms including multicore/manycore, FPGA and GP-GPU (graphics boards). It is also oriented toward intensive protein sequence comparison, and not for searching genomic banks. Then, its primarily goal is to compare two banks of protein sequences and to generate alignments where similar zones have been detected. The two protein banks are first loaded and indexed into the computer's main memory for optimizing the search process. However, for large banks, this process can take hours of computation. It is thus extremely interesting to evaluate the behavior of this algorithm onto parallel hardware.

During the last few years, there have been different parallel methods of sequence comparison algorithms. However, few implementations use manycore architecture, while manycore has been more and more popular at the same time. In this paper, we focus on a novel manycore architecture named Godson-T [4, 5], and show that the parallel part of the algorithm perfectly suits with the thread programming model of the Godson-T processor.

## 2. The Godson-T Architecture

As depicted in figure 1, a Godson-T chip has 24 computing tile nodes, 1 synchronization node, 4 IO controllers, and 4 memory controllers. The L2 cache is grouped into 4 banks. The 25 nodes are arranged in a 2D-mesh network, which provides both high bandwidth and fine scalability.

Figure 2 shows the architecture of one computing node: it consists of 4 processing cores, two L1 D-caches and one L1 I-cache. They are connected by a crossbar which provides a low-latency and high-bandwidth communication within each node. Besides the L1 cache, each node also has a SRAM bank which can be configured as a scratchpad memory (SPM) with very low access latency. SPM can be totally controlled by programmer. It can be used to speedup the access of frequently used data or to exploit the locality of some irregular computations.

978-0-7695-3443-5/08 $25.00 © 2008 IEEE
DOI 10.1109/PDCAT.2008.28



IEEE computer society

Authorized licensed use limited to: IEEE Xplore. Downloaded on January 26, 2009 at 10:04 from IEEE Xplore. Restrictions apply.

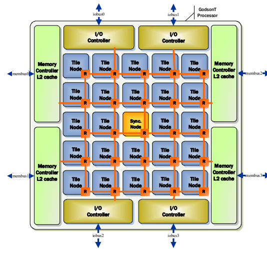



**Figure 1. Godson-T chip**

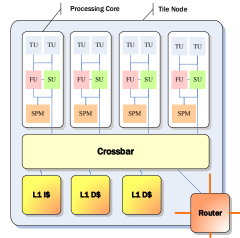


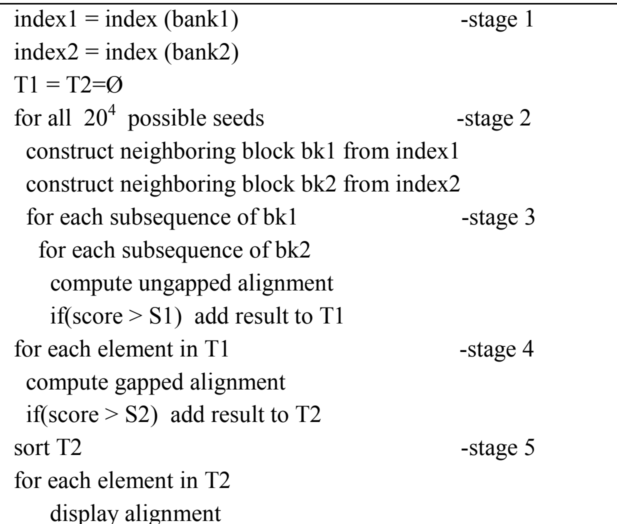

**Figure 2. Computing node**

In addition, Godson-T proposes a rich set of hardware to support synchronization. The way to use Acquire/Release (or Lock/Unlock) operations is similar to those based on shared memory access. With this mechanism, all the Acquire/Release operations can be recorded in the lock manager inside the chip. And this is much faster than traditional Acquire/Release operations which need additional memory accesses.

Godson-T uses a scope consistency (ScC) [6] memory model. Using the Godson-T hardware lock manager, we implement a lock-based cache coherence which is much simpler than snoopy-based or directory-based one. The L1 cache is write-through and the L2 cache is write-back. If it is the first time to read a shared variable after Acquire, Godson-T will always fetch it from L2 cache and then update the old value in L1 cache if there is an old copy there. This ensures the cache coherence among all the processing cores.

## 3. Protein Sequence Comparison Algorithm

This algorithm is mainly composed of 5 stages as shown in figure 3. Stage 1 makes indices for the two banks: each bank is stored into the computer's main memory using a specific data structure allowing fast retrieval of short words of length 4, called seeds. Stage 2 enumerates all the $20^4$ seeds (there are 20 different amino acids in a protein) and for each of them constructs 2 blocks of substrings: block bk1 contains substrings extracted from index1 and block bk2 contains substrings extracted from index2. A substring corresponds to neighboring amino acids around the seed, typically 40 amino acids. Stage 3 considers all possible pairs of substrings from bk1 and bk2 and starts the computation of an alignment by extending possible matches from the left and right hand side. If enough matches are found, then the extension is stored into structure T1. Stage 4 takes all elements of T1 and computes an alignment using dynamic programming techniques. Alignments exceeding a given threshold are stored in T2. Finally, stage 5 sorts the alignments before displaying them. The sort function aims at eliminating identical alignments found by stage 4.



```
index1 = index (bank1)                          -stage 1
index2 = index (bank2)
T1 = T2=Ø
for all 20^4 possible seeds                     -stage 2
  construct neighboring block bk1 from index1
  construct neighboring block bk2 from index2
  for each subsequence of bk1                   -stage 3
    for each subsequence of bk2
      compute ungapped alignment
      if(score > S1) add result to T1
for each element in T1                          -stage 4
  compute gapped alignment
  if(score > S2) add result to T2
sort T2                                         -stage 5
for each element in T2
    display alignment
```

**Figure 3. Sequential algorithm**

Normally, stage 3 and stage 4 dominate in terms of percentage of the execution time, especially when large banks are involved. When the number of sequences is more than 100 thousands, stage 3 and 4 get 64% and 33% respectively. Stage 5 gets 3%. The consuming time of stage 1 and 2 is almost negligible.

## 4. Parallelization and Performances

Clearly, an efficient parallelization has to target stage 3 and stage 4. Actually, for each seed, stage 2 to stage 4 can be processed independently. So we make each processor fetch a different seed and perform the corresponding stages (stage 2, 3, and 4).

Besides the serial components of the algorithm, there are 3 other factors which will affect the speed-up: (1) Synchronization overhead. This mainly comes from the Acquire/Release pairs used when the thread fetches a new word from the seed dictionary. We find that more than 90% of the entries in the seed dictionary are empty because there is no hit in both of the two banks. We'd better get rid of them first in order to avoid the unnecessary Acquire/Release pairs and decrease the lock competition among threads. (2) Memory contention. This algorithm is very cache-friendly thanks to the construction of neighboring blocks in stage 2. Different numbers of threads almost cause the same average percentage of time waiting for data access over the parallel part. So it is not a problem for our algorithm. (3) load balancing. This is very critical for the performance of parallelization, because the total parallel execution time is dependent on the longest



Authorized licensed use limited to: IEEE Xplore. Downloaded on January 26, 2009 at 10:04 from IEEE Xplore. Restrictions apply.

running thread. We find out that there is a good chance that matches of some special words will be bursty. There are always some words which have much more computation than others, and these words may lead to uneven computation in both stages 3 and 4.

```
Main thread
 index1 = index (bank1)                              -stage 1
 index2 = index (bank2)
 idx0=idx1=idx2=0;  T0 = T1 = Ø
 discard the redundant seeds and sort the others
 create P slave threads
 join salve threads
 merge alignment results from t1_p into T1
 sort T1                                             -stage 5
 for each element in T1
   display alignment result

Slave thread p
 while (seed < nb_seed)
   Acquire(lock0)
   iT0 = idx0; idx0++
   Release(lock0)
   get seed according to iT0
   construct neighboring block bk1 from index1       -stage 2
   construct neighboring block bk2 from index2
   t0_p = Ø;    t1_p = Ø
   for each subsequence of bk1                       -stage 3
     for each subsequence of bk2
       compute ungapped alignment
       if(score>S1)   add result to t0_p
   Acquire(lock1)
   iT0 = idx1; idx1 += size(t0_p)
   Release(lock1)
   store t0_p into T0 according to iT0
 Barrier
 while(idx2 < size(T0))                              -stage 4
   Acquire(lock2)
   iT0 = idx2; idx2+=K
   Release(lock2)
   Acquire(lock[p])
   get K ungapped results from T0 according to iT0
   Release(lock[p])
   for each ungapped result
     compute gapped alignment
     if (score > S2)   add alignment result to t1_p
```

**Figure 4. Parallel algorithm**

The parallel algorithm is depicted in figure 4. A main thread performs sequentially stages 1 and 5 while stages 2, 3 and 4 are parallelized on P slave threads.

The main thread starts by indexing the two banks. Then, it initializes 3 shared variables which will be used by the slave threads. P slave threads are then launched for executing the stages 2, 3 and 4. After that, the main thread waits for all slave threads to finish. Each slave thread returns a list of alignments which are merged into the T1 structure. The algorithm finishes similarly to the sequential algorithm: results are sorted before displayed.

Before performing stage 2, a slave thread has to get a unique seed number through the Acquire/Release pair. To minimize the Acquire/Release overhead, we discard all the redundant seeds in the main thread.

After getting a seed number, a slave thread constructs 2 neighboring blocks of substrings and performs ungapped extension. Results are stored in the internal structure t0_p. The next step is to move these results to the shared structure T0. Again, an Acquire/Release lock is used to reserve K places in T0, if K is the number of successful extensions to be stored. The idx1 variable is updated consequently.

In stage 4, to solve the load balancing problem, we re-divide all the elements in T0 among threads and make each thread get K elements from T0 dynamically. Before that, we add a barrier for the synchronization.

To be efficient, and to avoid long waiting time for the threads stuck in the barrier, we need to pay attention to the computation load of the loops in stage 3. As we mentioned before, seeds are not equivalent: some of them are largely overrepresented. If one of the last seeds belongs to this category, then the thread in charge of it will finish a long time after the others. To suppress this potential drawback, seeds are first sorted by complexity: seeds involving large computation time are computed first. In such a way, the last seeds require a very small amount of computation and the P threads end nearly at the same time. The overhead induces by the barrier synchronization is thus very small. Note that we have removed all the seeds with no hit before performing the sort operation. Thus the sort time is very small.

The second part of the computation deals with stage 4 for computing the final alignments. Each slave thread gets K extensions stored in T0 and processes independently K alignments. Getting the extensions must be controlled by an Acquire/Release pair. However, the same lock doesn't need to be used by all threads. We just want to make sure that the correct value can be extracted from the shared structure. Thus, different locks are used for different threads (lock[p] for thread p), and no thread has to wait.



Authorized licensed use limited to: IEEE Xplore. Downloaded on January 26, 2009 at 10:04 from IEEE Xplore. Restrictions apply.

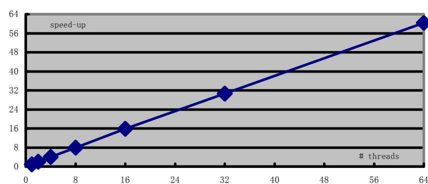



**Figure 5. Speed-up of parallel part**

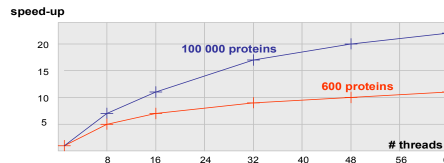



**Figure 6. Overall speed-up**

Performances have been estimated from a cycle-based event-driven simulator. Two banks of 300 protein sequences have been built. They are extracted from the SwissProt database (release 54.6) which has an average length of 360 amino acids.

The algorithm is simulated with an increasing number of threads, starting from 1 to 64. Each thread captures one computing node. We first measure the lock waiting time, which is the time spent in the various Acquire/Release pairs. The following table indicates the rate over the total parallel time:

| # ths | 2 | 4 | 8 | 16 | 32 | 64 |
|---|---|---|---|---|---|---|
| ‰ | 0.029 | 0.030 | 0.030 | 0.038 | 0.058 | 0.38 |

It can be seen that this time represents a very small fraction of the total parallel execution time. Similarly, we evaluate the average waiting time due to the synchronization barrier. For 64 threads, we get 0.63%; it is even smaller for a smaller number of threads. This small synchronization overhead has an immediate impact on the speed-up we can get from this parallelization. Figure 5 draws the speed-up of the parallel part as a function of the number of threads.

We nearly get a linear speed-up. This comes from the high efficiency of the Godson-T synchronization scheme and the fine scalability of the protein sequence comparison algorithm. If we now consider the global speed-up, including both the sequential part (stages 1 and 5) and the parallel part (stages 2, 3, and 4), we obtain the speed-up shown in figure 6.

The red curve (600 proteins) is the results of simulation of comparison between the two banks of 300 protein sequences. The blue one (100 000 proteins) is an extrapolation according to profiling. For small data set, stage 1 represents a non negligible percentage of the execution time (about 5%). For large data set, stage 1 becomes very small compared to the other stages, increasing the overall speed-up consequently. To get more speed-up, stage 5 need to be revisited in order to process data in a way suitable for parallelism.

## 5. Conclusion and Perspectives

Our results demonstrate that Godson-T is very efficient in the lock synchronization scheme. The total lock overhead is very small and can be almost negligible when using 64 threads. Our results also show that this algorithm has a fine scalability, and is very suitable to be parallelized. With a fine tuning for the synchronization and load balancing, we achieve a nearly linear speed-up when using 1 to 64 threads.

We only implement a coarse-grain parallelism, but it is also possible to use SIMD instructions to accelerate the most computation-intensive part: ungapped alignment. With SIMD support, each core can perform several ungapped alignments by using one instruction.

Note that in addition to protein sequences comparison, our implementation is also suitable for other forms of sequences comparison in bioinformatics.

## 6. Acknowledgements

We would like to thank INRIA which has funded our cooperation in IRISA. This work is also supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2005CB321600, and the National Natural Science Foundation of China under Grant No. 60736012.

## 7. References

[1] T.F. Smith, and M.S. Waterman, “Identification of common molecular subsequences”, *J Mol Biol*, 147(1), pp. 195-197. 1981

[2] S. Altschul, T. Madden, A. Schaffer, J. Zhang, Z. Zhang, W. Miller, D. Lipman, “Gapped BLAST and PSI-BLAST : A new generation of protein database search programs”, *Nucleic Acids Research*, Vol. 25, No. 17, pp. 3389-3402, 1997.

[3] W.R. Pearson, and D.J. Lipman, “Improved Tools for Biological Sequence Comparison”, *Proc. Natl. Acad. Sci.*, 85: 2444-2448. 1988

[4] G. Tan, D. Fan, J. Zhang, A. Russo, G. Gao, “Experience on Optimizing Irregular Computation for Memory Hierarchy in Manycore Architecture”, *PPoPP08*, Feb. 2008

[5] He Huang, Nan Yuan, Wei Lin .et al. “Architecture Supported Synchronization-Based Cache Coherence Protocol for Many-Core Processors”, *CMP-MSI’08, ISCA Workshop,* 2008

[6] L. Iftode, J. Singh, and K. Li, “Scope Consistency: A Bridge between Release Consistency and Entry Consistency”, *SPAA 96,* 1996



Authorized licensed use limited to: IEEE Xplore. Downloaded on January 26, 2009 at 10:04 from IEEE Xplore. Restrictions apply.