Noname manuscript No.
(will be inserted by the editor)

3D-Hit: fast structural comparison of proteins on multicore architectures

Ł. Bieniasz-Krzywiec, · M. Cytowski, · L. Rychlewski, · D. Plewczynski

Received: date / Accepted: date

Abstract 3D-Hit is well established method for rapid detection of structural similarities between proteins, which is widely used in various bioinformatics web servers (MetaServer, GRDB, 3D-Fun, Rosetta, etc.). The algorithm decomposes proteins into set of overlaping segments of 9-13 residues, then tries to match them using root mean square distance metric. The best aligned pairs of segments are selected as seeds for futher analysis. Those initial hits are expanded by iterative process in order to construct the global structural alignment by concatenating pairs of matching segments. The method has the same accuracy as the other state-of-the-art structural comparison algorithms (LGscore2, DALI), yet it provides much faster procesing times, and can be used in the high-throughput setup as the structural module in bioinformatics pipelines. The method is optimized in terms of speed and accuracy to work on novel computer architectures, such as PowerXCell8i and Sun Constellation System. Here, we provide the source code of the 3D-Hit program, describe selected architectures on which the software was ported, present programing models, point out significant porting steps and sumarize performance comparisons.

Keywords proteins · IBM Cell · structure comparison · bioinformatics · optimization

D. Plewczynski, Ł. Bieniasz-Krzywiec, M. Cytowski
Interdisciplinary Centre for Mathematical and Computational Modelling,
University of Warsaw (ICM),
Pawińskiego 5a, 02-106 Warsaw, Poland
E-mail: darman@icm.edu.pl

L. Rychlewski
BioInfoBank Institute,
Limanowskiego 24A16,
60-744 Poznań, Poland

1 Introduction

The PowerXCell8i is a pioneering microprocessor architecture is supposed to bridge the gap between conventional desktop computers and specialized high-performance machines. It has been designed to support very wide range of applications. The Cell processor consists of nine processor elements operating on a shared and coherent memory. Functions of those processors are specialized into two types. Cell has got one *PowerPC Processing Element* (PPE), which usually acts as a controller and is designed to run operating system. It also includes eight *Synergistic Processing Elements* (SPEs), which handle computational workload and are optimized for SIMD code execution. PowerXCell8i processors are usually available in a dual processor nodes within the IBM QS22 blades. This gives a shared memory programming environment with 16 computational cores. Moreover due to its specific architecture, Cell processor achieves very good performance results with relatively low power consumption (in terms of performance per Watt). As a result, *Nautilus* [1] — cluster of IBM QS22 blades based on Cell processors installed at *Interdisciplinary Centre for Mathematical and Computational Modelling (ICM)* has been ranked as No. 1 in two editions of The Green500 List [2] of the world most energy-efficient supercomputers.
The Sun Constellation System, another high-performance computer available at ICM, is a cluster based on a x86 architecture. This powerful machine consists of number of blades packed into special-purpose racks, which are tied together with highly-efficient InfiniBand connections. Each blade is equipped with four AMD Quad-Core Opteron 835X processors (which gives 16 computational cores per blade) and up to 32 Gb of memory.
In this paper we describe two fast implementations of an efficient scanning method for detecting structural similarities between proteins. The first of them is application designed for the PowerXCell8i processors. The second takes advantage of OpenMP and therefore can be executed on virtually all architectures providing shared memory access, including the Sun Constellation System. As a result we are able to compare two shared memory systems with different architectures but the same number of computational cores. Algorithm used in our programs was originally created by Dariusz Plewczyński et al. [3, 4]
The original code, destined to execute on x86 architecture, was ported using two different frameworks: Cell SDK with its SPE library (for PowerXCell8i architecture) and OpenMP (for parallel computers with shared memory). The Cell-accelerated application achieved an overall speedup of 12 over single threaded version executing on 1 SPE core. This level of performance was obtained with the use of all 16 SPU cores available within IBM QS22 blade. Program parallelized with OpenMP library performed even better in terms of the final walltime achieved during benchmarking tests. In the course of our work we encountered very interesting aspects of parallel programming and learned how to identity parts of code whose performance could benefit from novel high-performance architectures.

2 Structural alignment

It has been discovered that three dimensional structure of a protein is more conserved during the process of evolution than its sequence alone [5]. Therefore, the comparison of 3D structures of two proteins makes it possible to establish distant evolutionary relationships, even between very diverged proteins. As a result, the structural alignment of proteins increases our understanding of more distant evolutionary relationships [6, 7]. The correspondence between structural and functional classification enables scientists to determine functions of various newly discovered folds and whole protein families. Structural similarity can suggest evolutionary links between protein families, which can result in more detailed functional annotation of given protein on molecular level. Moreover, the structural comparison can guide the experimental structure determination process, by tracing shifts in low resolution models. The aforementioned reasons make the structural alignment the very important part of bioinformatics every-day work.
On the other hand, the size of Protein Data Bank [8] is growing rapidly doubling every 18 months. This huge amount of structural data needs very fast and accurate computer programs to deal with extracting structural information, and comparing new proteins with previously annotated ones. Those programs should enable not only structure-to-structure search, but also alignment over all proteins from the database in a real time. So far typical computations done has been performed by several state-of-the-art methods, including *3DHit* code. Because of the overwhelming and constantly growing amount of processing to be performed, scientists requested the support of the *Joint Cell Competence Centre* [10]. Due to an ongoing collaboration between *ICM* and *IBM*, it was decided to port the *3DHit* code, inter alia, to the PowerXCell8i Architecture and use Cell based machines as a computational facility.
The purpose of this article is to present how the *3DHit* program has been ported and tuned on novel architectures and how processing performance of accelerated versions compares to the x86 implementation.

3 Overview of the algorithm

3DHit program provides the structural alignment of two proteins. It uses in-house customised version of Smith-Waterman dynamic programing algorithm combined with intensive three-dimensional rotation and translation routines that align two geometrical objects in order to minimize the root mean square distance computed for all heavy atoms from both proteins. The pseudo-code of entire program is listed in the Algorithm 1 section. Virtually, the most time consuming part of the code is the preparation of structural alignment matrix, which is input data for the Smith-Waterman algorithm (step 2 of Algorithm 1). To identify similarity substructures of two proteins the program compares parts of their chains with a fixed length of 256 amino acids. We call them *segments*. Program analyses structural similarity between each pair of segments

as follows.
First of all it decides whether central parts of segments, which we call *seeds*, are similar enough to proceed with further computations. Seeds are very short subsequences of chains with the length of 13 amino acids. Algorithm makes rotations and translations of the Calpha atoms of both seeds in order to minimize the root mean square deviation (RMSD) between them. Secondly, if the structural similarity of the two seeds is high enough, the algorithm starts to analyze two longer continuous parts of the main chains centered on seeds. It uses a rotation matrix and a translation vector for a Cartesian-space superimposition of the two seeds to rotate and translate these large segments. Next, it defines the similarity matrix for the dynamic programming in the following way. If two Calpha atoms taken from the superimposed large segments are close enough in space (below 3A), it assigns 1 as their similarity score, 0 is taken otherwise. Then alignment score based on such similarity score matrix is computed. If it is high enough, algorithm passes this pair of segments on to the next filter. The whole procedure is repeated for the subsequences of length 100, 200 and 256 amino acids centered on seeds. The resulting score (number of superimposed Calpha atoms) is recorded in an additional final alignment matrix, containing segment superposition scores. This matrix is then used to find the best alignment of whole proteins. If no pair of segments has passed all filters, the overall score is 0.

Algorithm 1 A sketch of the structural alignment algorithm

Input: Two protein sequences: `protein1` and `protein2`.
Output: Structural alignment of the given proteins.

1. Divide each protein into segments of fixed length (256 amino acids).
2. `// Create a structural alignment matrix containing similarity`
   `// scores between each pair of segments.`
   For each `segment1` in `protein1` and `segment2` in `protein2`, compute the number of superimposed Calpha atoms in the aligned pair of segments:

```
similarityMatrix[segment1, segment2] =
    ComputeAlignmentOfSegments(segment1, segment2)
```

3. Find alignment of the whole proteins by running Smith-Waterman algorithm on them with the `similarityMatrix` as an input.

4 Porting process

After profiling the original version of *3DHit* program we found out that the most time consuming part of the code was a function implementing algorithm which compares two given segments. Intrinsically, it was executed in sequence with each pair of segments as a parameter. As a result, program spent more than 90% of its time in that function. We decided to accelerate that part of

Algorithm 2 `ComputeAlignmentOfSegments(segment1, segment2)`

Input: Two segments of potein sequences.
Output: The number of superimposed Calpha atoms in the aligned segments.

1. Extract the seed of each segment (seed is the central part of segment consisting of 13 amino acids).
2. Find the rotation matrix and the translation vector minimizing the RMSD between the two seeds.
3. If the optimal RMSD is too low, exit and return 0.
4. For `len` in {100, 200, 256}:
   (4a) Define the subsequences to consider:
   `subsequence1 =` the subsequence of length `len` surrounding the seed of `segment1`.
   `subsequence2 =` the subsequence of length `len` surrounding the seed of `segment2`.
   (4b) Align the two subsequences using the rotation matrix and the translation vector computed in the previous step.
   (4c) `// Define the alignment matrix for the two subsequences:`
   For each `atom1` in `subsequence1` and `atom2` in `subsequence2`:
   ```
   alignmentMatrix[atom1, atom2] =
       distance(atom1, atom2) < 3A ? 1 : 0
   ```
   (4d) Find the alignment of subsequences by running Smith-Waterman algorithm on them with the `alignmentMatrix` as an input.
   (4e) If computed alignment is better than the one found previously, continue the loop. Otherwise, return the best alignment found previously.

program by parallelization based on two programming models: libspe2 library model for Cell and OpenMP model for x86 architecture.

4.1 Parallel scheme

Computations for each pair of segments can be performed independently with no communication occurring between working threads. Nevertheless, at the end the main thread must collect all partial results for each pair of segments and compute final result based on them. We wondered whether those extra computations performed on the partial results would cause a new bottleneck. In the case of Cell implementation computing of the final result is handled by PPU. In the case of OpenMP version this is done by the master thread. Fortunately, it turned out in the end that accelerated versions of *3DHit* program spends only a fraction of percent of their execution times computing final result.

4.2 PowerXCell8i Implementation

We used the Cell SDK and its SPE library to implement this simple parallel scheme. The PPE processor executes the main thread of application. Its role is to read and preprocess description of two protein sequences, create SPE contexts, run appropriate working threads, collect partial results and finally

compute and return the final score and alignment. Each SPE processor gets its set of pairs of segments. Than, consecutively for each pair of segments from the set, it loads the descriptions of two segments into local memory space (Local Store), executes comparing function and stores the result back into the main memory.

4.2.1 Memory issue

Porting most of the applications to SPE processor is rather challenging task because of the limitation of size of Local Store, which is only 256 KB. This small space should be wisely administrated, because it must accommodate both program instructions and operating data. Moreover, the only way to load and store data to and from Local Store is DMA mechanism. It gives programmer a great deal of control, but on the other hand it is not simple to use. The *3DHit* code is unfortunately memory consuming. In the original version each execution of comparing algorithm requires memory for at least 256^2 floating point numbers and 256^2 short integer values. It is of course too much for the Local Store. That is why we had to perform some memory optimizations including compression and introduction of bit operations. The resulting code was slightly slower and less accurate than the original one, but it is an even trade. Thanks to those sacrifices we were able to fit our program to Local Store.

4.2.2 SIMD optimizations

According to the profile, the original version of *3DHit* program spends nearly 30% of its execution time calculating rotation matrices and translation vectors for Cartesian-space superimposition of pairs of segments. The part of code responsible for that task turned out to be a good candidate for a vectorization process due to the quantity of simple algebraic operations.
First of all, we decided to take advantage of `GCC` compiler auto SIMDizing abilities by switching on option `-ftree-vectorize`. At the beginning it did not help much since the code occurred to be to complex to automatic analysis for the compiler. Therefore we had to make some simplifications. We followed the guidelines proposed in [9].
Even after using specific compiler directives compiler was unfortunately unable to automatically vectorize some of the loops. We decided to tune the remaining parts of code manually by instruction substitution. The main vector operations used in the SPE computational kernel were `spu_add`, `spu_mul`, `spu_madd` and `spu_splats`. All of these instructions were operating on float 4 entry vectors, so we could speed up the vectorized loops of the appropriate steps of the algorithm about 4 times. The result of this effort is presented in Tab. 1.

Table 1 Performance results of SIMD and noSIMD versions.

Version	Compiler	Average time	Speedup
PowerXCell8i, noSIMD, 1 SPE	`GCC (-O3)`	1.546	1.0
PowerXCell8i, noSIMD, 16 SPEs	`GCC (-O3)`	0.181 s	8.541
PowerXCell8i, SIMD, 16 SPEs	`GCC (-O3 -ftree-vectorize)`	0.164 s	9.427

4.2.3 Efficient implementation of the parallel scheme

In our first approach to implement chosen parallel scheme we met very interesting problem. At the beginning we were assigning jobs for the computational kernels arbitrarily. Each SPE program was given a consecutive sequence of pairs of segments to operate on. Nevertheless, it was wrong decision because of the inefficient load balancing. As described before, comparing algorithm is sophisticated and it does not always behave in the same manner. For example, if it discovers that seeds are not similar enough, it finishes computations very quickly. Moreover, cutoffs may occur at the every stage, including analysis of longer segments. That is why it is important to assign each SPE with more or less the same amount of real work, which may not necessarily mean the same amount of pairs of segments.

We tried many different ways to fulfil this requirement. First of all, we decided to divide the whole set of tasks into 16 random subsets on the PPE side of application. Each SPE program operated on one of those randomly chosen kits. This solution on the one hand allowed us to achieve very good load balancing, but on the other hand drastically slowed down PPE thread. It took about 15% of its execution time to randomly permute set of tasks. We could not afford such a waste.

The next idea, which came to our minds, was to use PPE thread as a management resource distributing workload coherently with computations. In this approach each idle SPE asks PPE for a new commission. We implemented and tested a few versions taking advantage of SPE mailboxes as well as advanced DMA transfers with double buffering. Unfortunately non of them met our expectations, because of the slowdown caused by the communication process.

Finally, it turned out that the simplest solution is the best. Instead of randomly distributing the set of commissions on the PPE side, we have chosen an arbitrary random permutation and embed it into the SPE program as a constant. In this approach each SPE kernel is statically assigned a random set of tasks. It allowed us to completely eradicate communication between PPE and SPE and achieve reasonable load balancing. The performance comparison of all of these methods is presented in Tab. 2.

4.2.4 Other optimizations

Each SPE program is executed once at the beginning and serves as a computational facility for many tasks. Thanks to that we are able to eliminate time needed for SPE context creation. In addition we decided to make use of

Table 2 Performance results of various versions.

Version	Compiler	Average time
Randomization on PPE, 16 SPEs	`GCC (-O3 -ftree-vectorize)`	0.247 s
Dynamic distribution, 16 SPEs	`GCC (-O3 -ftree-vectorize)`	0.370 s
Static permutation, 16 SPEs	`GCC (-O3 -ftree-vectorize)`	0.164 s

DMA double buffering mechanism. While SPE is carrying on computations, its Memory Controller can coherently load the next portions of data from the main memory. This simple idea let us hide input-output operations behind real computations.
In comparison to the single SPE, we achieved an overall speedup of 6.14 while executing on 8 SPEs and 9.39 while executing on 16 SPEs. We decided to put into use one more improvement. We run two parallel instances of *3DHit* program, each using 8 SPEs, on the QS22 server equipped with two Cell chips. That gives as an average speedup of 12.

4.3 OpenMP Implementation

Implementing chosen programming model using OpenMP occurred to be very simple. The whole set of pairs of segments is dynamically (and automatically) distributed among number of OpenMP threads. Each such a thread executes an algorithm similar to the one described in the PowerXCell8i's section. It gets a pair of segments, computes an answer and stores the result in the specified place in the main memory.
In spite of its simplicity, the OpenMP implementation of our application occurred to be very efficient and accurate. In comparison to the single-threaded version, we achieved an overall speedup of 4.37 while executing on 8 cores. We take advantage of the same schema as the one used with PowerXCell8i processors. We run two parallel instances of *3DHit* program, each using 8 threads, on blade equipped with 16 processor units. That gives as an average speedup of 8.5 over the initial x86 version.

5 Performance results

5.1 PowerXCell8i code analysis

We have used a `spu_timing` facility to analyze and tune computational kernels of *3DHit* Cell implementation. Results presented in Tab. 3 show that we achieved very good scaling over increasing number of working SPE threads. The part of code, which can be parallelized, takes about 97.994% of execution time of single-threaded version of application. Therefore, according to the Amdahl's Law, the best theoretical speedup with 16 working threads, is approximately $\frac{1}{(1-0.97994)+\frac{0.97994}{16}} \approx 12$. Our result is slightly worse. We achieve

an overall speedup of almost 9. The most probable cause is using built-in permutation of commissions instead of actually randomly permutated set of segments. This approach increases probability of occurrence of inefficient load balancing.

Table 3 Profile results.

Part of code	**1 SPE**	**8 SPEs**	**16 SPEs**
Preparing data	0.004782 %	0.142613 %	0.359532 %
Creating SPE threads	0.121318 %	7.627334 %	21.029137 %
Waiting for threads	97.844704 %	80.333420 %	62.397312 %
Computing global result	0.019680 %	0.139842 %	0.219315 %

Part of code	**1 SPE**	**8 SPEs**	**16 SPEs**
Preparing data	0.000051 s	0.000247 s	0.000466 s
Creating SPE threads	0.001170 s	0.012919 s	0.027042 s
Waiting for threads	1.492583 s	0.207126 s	0.114413 s
Computing global result	0.000527 s	0.000571 s	0.000619 s

5.2 Performance Comparison

We have designed test to examine operational performance of *3DHit* code executing on various architectures. We have chosen a set of 18 proteins from a database and compared execution times on an Quad-Core AMD Opteron Processor 8354 based nodes and QS22 server. Each pair of proteins from the test set was compared during a test run, which gave us 324 single test cases. The average results are presented in Tab. 4. As we can see, Cell accelerated version

Table 4 Performance results on two systems: AMD Opteron 8354 and PowerXCell8i QS22.

Architecture	**Compiler**	**Time**	**Speedup**
AMD, 1 OpenMP thread	`GCC (-O3)`	116.85 s	1.00
AMD, 4 OpenMP threads	`GCC (-O3)`	38.22 s	3.05
AMD, 8 OpenMP threads	`GCC (-O3)`	26.72 s	4.37
AMD, 16 OpenMP threads	`GCC (-O3)`	26.09 s	4.47
PowerXCell8i, 1 SPE	`GCC (-O3 -ftree-vectorize)`	499.07 s	1.00
PowerXCell8i, 8 SPEs	`GCC (-O3 -ftree-vectorize)`	81.20 s	6.14
PowerXCell8i, 16 SPEs	`GCC (-O3 -ftree-vectorize)`	53.14 s	9.39

of *3DHit* occurred to be approximately two times slower than the OpenMP code executed on blades equipped with AMD processors. Those performance differences could be caused by disparities in technical parameters of chosen

machines. According to our experiments, multiplication of two floating point scalars is almost two times slower on PowerXCell8i SPE processors than on AMD Opterons. Unfortunately a great deal of *3DHit* code could not be vectorized and, as a result, arithmetic operations on scalars are intense in our code. Moreover, the PowerXCell8i version is slightly more complex. For example, necessity of performing memory optimisations, described in one of previous sections, has its own overhead. Another very important feature that kills the performance of the implemented SPU kernels is a big number of branches introduced by the algorithm and a lack of branch prediction mechanism within the SPU architecture. As a result one of the most important computational parts of the code, Smith-Waterman algorithm, is significantly slowed down on Cell architecture. Taken together, all those factors cause the drop-down of performance.
On the other hand, PowerXCell8i implementation has one desired property, which is unfortunately not a feature of OpenMP-based program, namely — very good scalability. It is probably due to the limited memory bandwidth. When 16 working threads try to simultaneously read data from shared memory located on their blade, they experience significant slowdown of data transfer. It should be stated here that the comparison of two longest sequences in the benchmark test takes approximately only 0.22 seconds. The memory bandwidth is not a problem in PowerXCell8i implementation, due to the presence of highly efficient Element Interconnect Bus (EIB), which provides each SPE and its memory controller with private and very fast connection to the main memory.

6 Summary

The current evaluation of *3DHit* performed on dataset of circa 300 query proteins reveals the quality of the tool, as compared with other programs. When compared with DALI server, our tool is able to generate similar number of correct models, however the final alignment quality is better in the case of the second service. In the case of distant structural comparisons, our method gets better ratings in all categories (such as specificity analysis) when MaxSub evaluation method is used. Concluding, the *3DHit* software is on average less sensitive than the DALI server, yet it is better than CE or VAST tools.
On the side of optimizalization of the core, we have ported the *3DHit* program to the novel high-performance architectures and achieved very good speedup. Our accelerated programs are planed to be embedded into the web application, which will allow very fast and accurate mechanism for structural alignment of proteins. By taking advantage of PowerXCell8i and Quad-Core x86 novel architectures, we were able to significantly reduce time of single computation and as a result provide scientists all over the world with always up-to-date and fast tool for structural alignment experiments. The present version of our algorithm is very fast. It takes circa 2 seconds for a query protein of 500 amino acids to scan a database of 1000 templates. The single comparison of two pro-

teins with size circa 500 amino acids is performed within a runtime of 0.002 seconds for *3DHit*, where older version of the software took around 0.017 seconds. Other structural alignment programs are significantly slower, CE takes 3 seconds to compare two typical proteins, LGScore2 is around 6 seconds.
Because of its speed and portability (the source code is available from authors upon request) we believe that *3DHit* program will continue to be widely used in structural genomics projects, improving the structural comparison of newly crystallized proteins with large structural databases. We are planning to provide the internet web server interface to the PDB database, in order to give user access to rapid structural alignment of its protein of interest with three-dimensional crystals or 3D models of proteins.

7 Acknowledgments

This work was supported by EC OxyGreen (KBBE-2007-212281) 6FP project as well as the Polish Ministry of Education and Science (N301 159735, and others).

References

[1] Nautilus system, http://cell.icm.edu.pl/index.php/Nautilus
[2] The Green500 List, http://www.green500.org/lists/2009/06/list.php
[3] Plewczynski D, Pas J, von Grotthuss M, Rychlewski L. *3D-Hit: fast structural comparison of proteins.* Appl Bioinformatics. 2002;1(4):223-5.
[4] Plewczynski D, Pas J, Von Grotthuss M, Rychlewski L. *Comparison of proteins based on segments structural similarity.* Acta Biochim Pol. 2004;51(1):161-72.
[5] Chothia C, Lesk AM. *The relation between the divergence of sequence and structure in proteins.* EMBO J.; 5: 823-6 (1986).
[6] Bujnicki JM. *Phylogeny of the restriction endonuclease-like superfamily inferred from comparison of protein structures.* J Mol Evol.; 50: 39-44 (2000).
[7] Johnson MS, Sutcliffe MJ, Blundell TL. *Molecular anatomy: Phyletic relationships derived from three-dimensional structures of proteins.* J Mol Evol.; 30: 43-59 (1990).
[8] Berman HM, Westbrook J, Feng Z, Gilliland G, Bhat TN, Weissig H, Shindyalov IN, Bourne PE. *The Protein Data Bank.* Nucleic Acids Res.; 28: 235-42 (2000).
[9] Abraham Arevalo, Ricardo M. Matinata, Maharaja Pandian, Eitan Peri, Kurtis Ruby, Francois Thomas, Chris Almond, *Programming the PowerXCell8i Architecture, Examples and Best Practices*, RedBooks, IBM (2008)
[10] IBM, ICM (University of Warsaw): Joint Cell Competence Center, http://cell.icm.edu.pl