Solving the Motif Finding Problem on a Heterogeneous Cluster using CPUs, GPUs, and MIC Architectures

H. M. Faheem[ǂ*ψ], B. Koenig-Riez[*], Mahmoud Fayez[ǂψ], Iyad Katib[¥], and N. Al-Johani[¥]

ǂ Ain Shams University, * Jena University, ψ Fujitsu, ¥ King Abdul Aziz University

Abstract - **The Motif Finding Problem MFP is a computationally intensive problem in the bioinformatics domain. Solving such problems on heterogeneous clusters consisting of CPUs, CUDA GPUs, and Intel Many Core (MIC) architectures is considered a challenging problem. This paper solves the MFP on a heterogeneous cluster using a scheduling strategy intended to schedule tasks on heterogeneous architectures based on their speed. The main idea is to solve the problem using suitable parallel computing paradigms such as MPI, OpenMP, and CUDA on individual architectures then to estimate the number of tasks that should be assigned to each one based on its speed in solving such tasks. We can find that the total execution time will be significantly improved when compared to pure CPU-based implementation. Of course this significant improvement will be obvious when we have relatively compared numbers of nodes of different architectures. The paper also shows that the speedup is inversely proportional to the increased number of CPUs since excessive number of CPUs can eliminate the effect of using faster architectures. However, using excessive number of CPUs in only one job to achieve considerable speedup factor has a great impact on the system utilization and consequently on the concurrent number of jobs that can be submitted to the system. The paper then shows how to modify the code to assign the tasks to the architectures.**

Keywords- Heterogeneous Architectures, Motif Finding Problem, Task Scheduling.

I. Introduction

Modern high performance computing (HPC) clusters have traditional multicore microprocessors (CPUs), graphics processor units (GPUs), and Intel many integrated core (MIC) architectures. This in turn leads to more heterogeneity among the computational resources within a single cluster. Writing an efficient code that can optimally utilize these heterogeneous resources depends mainly on the capabilities of the developer and the parallel computing paradigm he uses. Moreover, the scheduling strategy dealing with such heterogeneity is considered one of the most important factors affecting the performance of the heterogeneous systems.

In this paper, we will solve one of the computationally intensive problems in the bioinformatics field. The problem is called “Motif Finding Problem”. We will use brute force algorithm to solve this problem three times. The first will be on multicore CPUs, while the second will be on GPU, and the third will be on MIC. Consequently, the actual run time for each will be calculated. Eventually, we will use a specific scheduling strategy to assign proper workload to the architectures to achieve near optimal hardware resource utilization and more speedup. We will also see how to modify the code to cope with the deployed scheduling strategy.

The rest of this paper is organized as follows: section II describes the motif finding problem. Section III shows the implementation on different architectures. Section IV briefly explains the deployed scheduling strategy. Section V presents the changes to the parallel code to fulfill the scheduling strategy requirements. Section VI contains some concluding remarks and directions for future work.

II. Motif Finding Problem

The Motif Finding Problem (MFP) can be simply considered as a string matching problem. Solving the MFP to find a motif of length *L* with permitted mutation *d* can be implemented using a brute-force algorithm. All the possible *L-mers* (4^L) are compared with each possible motif of length *L*. If we have a sequence of size *N* then we can have (*N-L+1*) motifs. Pevzner and Sze [1] presented the challenge problem (15, 4) where the first number is a specific length *L* and the second number a specific mutation *d*. In this paper, we present a problem in which the motif has a length *L*=15, allowed mutations *d*=4, and the number of sequences we are searching in is *T*=20 each of size *N*=600. Solving such computationally intensive problems can be implemented using a set of heterogeneous platforms [2, 3, 4, 5, 6, 7, 8, 9]. Possible characters to construct a DNA sequences is represented in the regular expression shown in (1). Possible L_{mers} of length *L* is represented in the regular expression shown in (2). Set of sequences is represented in (3). The function *match* is used to compare two motifs *A* and *B* each of size *L* is shown in (4) where A_i and B_i represent the i^{th} position into the *A* and *B* motifs. The function *score* is responsible for counting the existence of a specific *L-mer* in all the *T* sequences as in (5). The motif of maximum occurrence is denoted as *motif* and is shown in (6).

$$V \rightarrow \mathbf{A}|\mathbf{C}|\mathbf{G}|\mathbf{T} \qquad (1)$$

$$Possible\ L_{mers} \rightarrow V^l \qquad (2)$$

$$S = \{s_1, s_2, \dots, s_T\} \qquad (3)$$

III. Implementing MFP on different Architectures

In this section we will show the results of implementing the MFP on different architectures. All the experiments on the system have been implemented using Intel Compiler 2015 and Intel MPI V5. GPU experiments implemented using CUDA V6 and GCC compiler and OpenMPI. MIC experiments are implemented using Intel Compiler 2015 and Intel MPI V5 using native mode for MIC. Table 1, 2, and 3 show the typical architectures of regular CPU node, MIC (Xeon Phi) node, and NVIDIA GPGPU (CUDA) node respectively. Infiniband network is used to connect different compute nodes.

Table 4 shows the scalability of the problem when using different number of CPU cores and MPI and OpenMP parallel computing paradigms. We can see that as the number of cores increases the time needed to find a solution considerably decreases.

$$match(A,B,l,d) = \begin{cases} 1, & l-d \geq \sum_i \begin{cases} 1, & A_i = B_i \\ 0, & else \end{cases} \\ 0, & else \end{cases} \qquad (4)$$

$$score(L\_mer, S, d) = \sum_{i=1}^{T} \sum_{k=0}^{N-l+1} match(L\_mer, s_i[k, \dots, k+l], l, d) \qquad (5)$$

$$motif = \{m \mid m = MAX\ (score(L\_mer, S, m)\ \forall\ L\_mer\ \in Possible\ L_{mers})\} \qquad (6)$$

Table 1: Regular CPU-based Compute Node

Attribute	Value
Architecture	x86_64
CPU op-mode(s)	32-bit, 64-bit
Byte Order	Little Endian
CPU(s)	24
On-line CPU(s) list	0-23
Thread(s) per core	1
Core(s) per socket	12
Socket(s)	2
NUMA node(s)	2
CPU MHz	2399.852
Memory	96 GB

Table 2: Xeon Phi Compute Node

Attribute	Value
Total No of Active Cores	60
Voltage	897000 uV
Frequency	1052631 kHz

IV. Scheduling Strategy

In this section we will examine the impact of using scheduling strategy described in [10] on assigning tasks to different architectures. Consequently, we will show how the code will be affected based on such scheduling strategy. The objective of this scheduling strategy is to minimize the time needed to solve the MFP. Since we have 4^{15} tasks, each task will compare one *L-mer* with all the possible windows extracted from all the given sequences, hence the total number of comparison operations in each task *CMP* is described in (7) while the total number of comparison operations for all tasks CMP_t is shown in (8). Table 5 shows the speed differences between architectures when running one task. We simply run the task separately on the architecture and calculate the execution time. This in fact can give us a ratio on which we can decide how many tasks could be assigned to a specific architecture. Specific ratio of tasks that should be assigned to different architectures from a total number of (4^L tasks) is listed in table 6. Modified run times are also shown in this table.

Table 3: NVIDIA CUDA Compute Node

Attribute	Value
CUDA Driver Version / Runtime Version	6.0 / 6.0
CUDA Capability Major/Minor version number	3.5
Total amount of global memory	5120 MBytes (5368512512 bytes)
(13) Multiprocessors, (192) CUDA Cores/MP	2496 CUDA Cores
GPU Clock rate	706 MHz (0.71 GHz)
Memory Clock rate	2600 Mhz
Memory Bus Width	320-bit
L2 Cache Size	1310720 bytes
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	49152 bytes
Total number of registers available per block	65536
Warp size	32
Maximum number of threads per multiprocessor	2048
Maximum number of threads per block	1024
Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)
Max dimension size of a grid size (x,y,z)	(2147483647, 65535, 65535)
Maximum memory pitch	2147483647 bytes

Having a look to the results in tables 4 and 6 can give us an idea about the improvement in the total run time. For example; implementing the brute force algorithm using one regular node, one CUDA node, and one Xeon Phi node will reduce the run time from 13373 seconds on a single regular node to 2333 seconds with a speedup factor of 5.7 while using four regular nodes, one CUDA node, and one Xeon Phi node will reduce the run time from 3353 seconds on pure 4 regular nodes to 1533 seconds with a speedup factor of 2.18. Since the number of CUDA nodes and Xeon Phi nodes are fixed in our cluster then we can find that as the number of regular nodes increases, the speedup factor decreases.

$$CMP = (N - L + 1) * T \tag{7}$$

$$CMP_T = 4^L * CMP \tag{8}$$

Table 4: Implementation Results of solving MFP on CPUs, Xeon Phi, and NVIDIA CUDA

Trial No.	Platform	Result (seconds)
1	1 Regular Node (OpenMP)	13373
2	1 Regular Node (MPI + OpenMP)	13263
3	2 Regular Nodes (MPI + OpenMP)	6590
4	4 Regular Nodes (MPI + OpenMP)	3353
5	8 Regular Nodes (MPI + OpenMP)	1688
6	16 Regular Nodes (MPI + OpenMP)	851
7	32 Regular Nodes (MPI + OpenMP)	430
8	64 Regular Nodes (MPI + OpenMP)	216
9	128 Regular Nodes (MPI + OpenMP)	109
10	256 Regular Nodes (MPI + OpenMP)	56
11	1 XEON Phi Node (Native Mode + OpenMP)	22446
12	1 GPU Node (CUDA)	3234

Table 5: Speed differences of the architectures to complete one task

Architecture	CPU Node	GPU Node	XEON-Phi Node
Task Execution Time (in Sec.)	1.24E-05	3.01E-06	2.09E-05

Table 6: Tasks assigned to architectures based on their speeds

Platform	**CPU Ratio %**	**CUDA Ratio %**	**MIC Ratio %**	**Result (seconds)**
1 Regular Node+ 1 CUDA+1 MIC	17.44923	72.15477	10.39600	2333.49
1 Regular Node+ 1 CUDA+1 MIC	17.56852	72.05050	10.38097	2330.11
2 Regular Nodes+ 1 CUDA+1 MIC	30.01815	61.16871	8.81313	1978.20
4 Regular Nodes+ 1 CUDA + 1 MIC	45.74194	47.42509	6.83297	1533.72
8 Regular Nodes + 1 CUDA+1 MIC	62.61125	32.68021	4.70854	1056.87
16 Regular Nodes+ 1 CUDA+1 MIC	76.86071	20.22525	2.91404	654.08
32 Regular Nodes+ 1 CUDA+ 1 MIC	86.79656	11.54067	1.66277	373.23
64 Regular Nodes+ 1 CUDA+1 MIC	92.90111	6.20490	0.89400	200.66
128 Regular Nodes+ 1 CUDA+1 MIC	96.28712	3.24530	0.46758	104.95
256 Regular Nodes+ 1 CUDA+1 MIC	98.05740	1.69796	0.24464	54.91

V. Modifications to the Code

The main program ***MotifTaskScheduler*** is responsible for dividing the problem space into smaller subspaces called chunks. Each chunk consists of a set of tasks. The chunk will run on the corresponding architecture. If certain architecture cannot process a single task in a reasonable time which must be less than the time for the fastest architecture to process all the tasks; then this architecture must be ignored. Fig.1 shows the pseudo code used to schedule the MFP on the different architectures. The code is designed to get the available MFP implementations from the database. The code then checks if the required architecture(s) is online for each algorithm or not. The list of algorithms that can be implemented on the on-line architectures will have a pre-calculated ratio stored into the database. This ratio is generated by the scheduling strategy. These ratios should be normalized before dividing the chunks. This step is necessary in case one of the algorithms cannot run in the current state of the system due to unavailability of the required architecture. The code then determines the *start* and *end* indices for each architecture to process. Eventually, the workload is distributed among different architectures to process a specific range using a specific algorithm.

The pseudo code shown in Fig. 2 represents the OpenMP implementation for finding the motif on CPUs. Given the *Start* and *End* indices of the *L-mers*, the MFP_OMP subroutine compares each *L-mer* in the range with all the possible *CMP* windows (extracted from *T* sequences, each of *N* characters) of the same size *L* and then records the score of each *L-mer*. The motif of the highest score ($score_{max}$) will be registered.

The Pseudo code of MPI implementation for finding the motif is listed in Fig. 3. Similar to MFP_OMP subroutine the MFP_MPI subroutine divides the search space and each MPI rank uses the MFP_OMP routine to search its subspace. The root rank (rank 0) will collect and apply maximum reduction offered by MPI to find the global Motif.

The pseudo code shown in Fig. 4 is similar to that of OpenMP except the replacement of OpenMP directive with the Offload directive to run this block of code on the MIC co-processor.

```
1.  PROGRAM MotifTaskScheduler
2.  Input : S[1, …, T]
3.  Input : L
4.  Input : d
5.  BEGIN
6.  $t[t_1, \dots, t_p] \leftarrow$ load single task execution time for architectures
7.  $t_{min} \leftarrow MIN_{i=1}^{p}(t_i) * 4^L$ ; find the smallest run time
8.  FOR $i = 1$ to $p$
9.      IF $t_i \le t_{min}$ THEN
10.         $R_i \leftarrow 4^L / t_i$   ; find the weight of each architecture
11.     ELSE
12.         $R_i \leftarrow 0$ ; this architecture is very slow and will be ignored
13.     END
14. END
15. $R_{total} \leftarrow R_1 + R_2 + \cdots + R_p$ ; sum the weights
16. $R_u \leftarrow 4^L / R_{total}$ ; find the tasks assigned to each weight unit
    $offset = 0$
    $start_i = 0$
17. FOR $i = 1$ to $p$
18.     $C_i = R_i * R_u$ ;tasks assigned to architecture
        $start_i = start_i + offset$ ; determine the start index of tasks
        $end_i = start_i + C_i - 1$ ; determine the end index of tasks
        $offset = C_i$
19.     $Score_i \leftarrow$ SPAWN $Algorithm_i(S, L, d, C_i, start_i, end_i)$
20. END
21. return $MAX_{i=1}^{p}(Score_i)$ ; find the motif of highest occurrence
22. END
```

Fig. 1: Scheduling routine pseudo code to assign workloads to architectures.

Fig. 5 shows the host side pseudo code for the CUDA implementation. This module calculates the number of CUDA Blocks and activates sufficient number of threads per block. Fig. 6 shows the actual kernel function that will perform the matching process. The CUDA kernel is called 8 times as it takes very long time to process a single job which causes a timeout so we divided the search space into 8 chunks of size (512*Blocks) which is a 2D job. The thread index is considered the Motif that must be matched against all the *CMP* windows. As we have multiple batches for the job, an offset is applied to the thread index to differentiate the chunks. The host side starts the CUDA jobs and collects the results in a global array. Finally the host finds the highest score in the global array.

VI. Conclusion

Solving computationally intensive problems on heterogeneous architectures can significantly improve and speedup the run time of the problem solution when proper scheduling strategy and suitable parallel computing paradigms are used. Having equivalent or at least comparable number of different architectures can result in a tangible speedup. Deploying more and more CPUs can bridge the gap of speed difference between architectures but will result in fewer number of concurrent jobs that can be allocated to the system. This is due to the increased percentage of utilized resources. Future work may include the use of different scheduling strategies and intelligent selection criteria to choose the best scheduling strategy to solve a given computationally intensive problem. We also may investigate the power consumption since we believe that deploying an excessive number of CPU-based nodes can result in excessive power consumption. We believe that this paper is a step towards a complete system to solve computationally intensive problems on heterogeneous architectures.

```
1.  PROGRAM MFP_OMP
2.  Input : $subspace(start, \dots, end)$
3.  Input : $S[1, \dots, T]$
4.  BEGIN
5.  $motif \leftarrow 0$
6.  $score_{max} \leftarrow 0$
7.  $OpenMP Directive
8.  FOR ALL motif $x$ in $subspace[start, \dots, end]$
9.      BEGIN
10.         $score \leftarrow score(x, S)$
11.         IF $score > score_{max}$ THEN
12.             $motif \leftarrow x$
13.             $score_{max} \leftarrow score$
14.         ENDIF
15.     END
16. return $score_{max}$
17. END
```

Fig. 2 Pseudo code for OpenMP implementation for MFP

```
1.  PROGRAM MFP_MPI
2.  Input : $subspace(start, \dots, end)$
3.  Input : $S[1, \dots, T]$
4.  BEGIN
5.  $motif \leftarrow 0$
6.  $score_{max} \leftarrow 0$
7.  $rank \leftarrow MPI\ Rank$
8.  $mpi_{start} \leftarrow rank * ((end - start) / MPI\_Size)$
9.  $mpi_{end} \leftarrow (rank + 1) * ((end - start) / MPI\_Size)$
10. Call MFP_OMP($subspace[mpi_{start}, \dots, mpi_{end}]$, S)
16. return $MPI\_Reduction(MAX, score_{max})$
17. END
```

Fig. 3 Pseudo code MPI implementation for MFP

VII. Acknowledgement

Computation for the work described in this paper was supported by King Abdulaziz University's High Performance Computing Center (Aziz Supercomputer) (http://hpc.kau.edu.sa).

```
1.  PROGRAM MFP_MIC
2.  Input : $subspace(start, \dots, end)$
3.  Input : $S[1, \dots, T]$
4.  BEGIN
5.  $motif \leftarrow 0$
6.  $score_{max} \leftarrow 0$
7.  $OFFLOAD Directive
8.  FOR ALL $motif\ x\ in\ subspace[start, \dots, end]$
9.      BEGIN
10.         $score \leftarrow score\ (x, S)$
11.         IF $score > score_{max}$ THEN
12.             $motif \leftarrow x$
13.             $score_{max} \leftarrow score$
14.         ENDIF
15.     END
16. return $score_{max}$
17. END
```

Fig. 4: Pseudo code for MIC implementation for MFP

```
1.  PROGRAM MFP_GPU
2.  Input : $subspace(start, \dots, end)$
3.  Input : $S[1, \dots, T]$
4.  BEGIN
5.  $threads \leftarrow 512$
6.  $blocks \leftarrow \mathbf{ceiling}(\ (end - start)\ /\ 8\ \ /\ threads)$
7.  $score[start, \dots, end] \leftarrow 0$
8.  FOR n 0 to 7
9.      BEGIN
10.         Call MotifKernel<blocks, threads>(S, n * (end-start)/8 , score)
11.     END
13. return $MAX_{i=start}^{end}(score_i)$
14. END
```

Fig. 5: Pseudo code for CUDA implementation for MFP

```
1   Module MotifKernel
2   Input : $S[1, \dots, T]$
3   Input : $offset$
4.  Output : $score[start, \dots, end]$
5.  BEGIN
6.  $thread \leftarrow (blockIdx * 256 + threadIdx) + offset$
7.  $score[thread] \leftarrow score\ (thread, S)$
8.  END
```

Fig. 6 Pseudo code for CUDA kernel function that runs on the GPU

References

[1] P. Pevzner and S. Sze, " Combinatorial approaches to finding subtle signals in DNA sequences," Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology, 269–78, 2000.

[2] H. M. Faheem, "Accelerating Motif Finding Problem using Grid Computing with Enhanced Brute Force," The 12th International Conference on Advanced Communication Technology (ICACT), Korea, 2010.

[3] M. Raddad, N. El-Fishawi, and H. M. Faheem, “Implementation of Recursive Brute Force for Solving Motif Finding Problem on Multi-Core,” International Journal of Systems Biology and Biomedical Technologies, 2 (3):1-18, 2013.

[4] R. Inta and D. J. Bowman, “An FPGA/GPU/CPU hybrid platform for solving hard computational problems,” in Proceedings of the eResearch Australasia, Gold Coast, Australia, 2010.

[5] S. J. Park, D. R. Shires, and B. J. Henz, “Coprocessor computing with FPGA and GPU,” in Proceedings of the Department of Defense High Performance ComputingModernization Program:Users Group Conference—Solving the Hard Problems, pp. 366–370, Seattle,Wash, USA, 2008.

[6] M. Showerman, J. Enos, A. Pant et al., “QP: a heterogeneous multi-accelerator cluster,” in Proceedings of the 10th LCI International Conference on High-Performance Cluster Computing, vol. 7800, pp. 1–8, Boulder, Colo, USA, 2009.

[7] D. B. Thomas, L. Howes, andW. Luk, “A comparison of CPUs, GPUs, FPGAs, and massively processor arrays for random number generation,” in Proceedings of the 7th ACM SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA ’09), pp. 63–72, Monterey, Calif, USA, 2009.

[8] K. Underwood, “FPGAs vs. CPUs: trends in peak floatingpoint performance,” in Proceedings of the 12th International Symposium on Field-Programmable Gate Arrays (FPGA ’04), pp. 171–180, New York, NY, USA, February 2004.

[9] H. Khaled, H. M. Faheem, and R. El-Gohary, “Design and implementation of a hybrid MPI-CUDA model for the Smith-Waterman algorithm,“ International Journal of Data Mining and Bioinformatics, 12(3): 313-327, Inderscience, 2015.

[10] H. M. Faheem and B. König-Ries, “A New Scheduling Strategy for Solving the Motif Finding Problem on Heterogeneous Architectures,” International Journal of Computer Applications, 101(5):27-31, 2014