2013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum

Accelerating all-to-all protein structures comparison with TMalign using a NoC many-cores processor architecture

Anuj Sharma*, Antonis Papanikolaou†, Elias S. Manolakos*

*Department of Informatics and Telecommunications
University of Athens, Athens, Greece
Email: {asharma, eliasm}@di.uoa.gr

†Institute of Communication and Computer Systems
National Technical University of Athens, Athens, Greece
Email: antonis@microlab.ntua.gr

***Abstract*—Computational challenges for the one-to-many and many-to-many protein structure comparison (PSC) problem are a result of several factors: constantly expanding large-size structural proteomics databases, high computational complexity of pairwise protein comparison algorithms, and the multitude of pairwise comparison approaches used in the field. Advances in processor architectures, such as many-core CPUs, have enabled them to support parallelism making them of interest in speeding up PSC techniques. We present rckAlign, an implementation of the popularly used TM-align PSC algorithm, designed for the Single-Chip Cloud Computer (SCC), an experimental processor created by Intel Labs. We developed a skeleton library, rckskel, and implemented a master-slaves variant of TM-align to exploit the parallelism offered by the SCC. We evaluated rckAlign on the SCC and compared it with existing TM-align software running on a dual-core AMD CPU (2.4 GHz) and on a single-core Intel P54C Pentium CPU (800 MHz). We observed an 11-fold speedup relatively to the former and a 44-fold speedup relatively to the latter. A key aspect of the performance of rckAlign on the SCC, is the almost linear speedup achieved with the number of SCC cores used as slaves. The method presented can easily be applied to other PSC algorithms and extended to running multiple PSC algorithms within the same SCC chip.**

***Keywords*-Protein structure comparison, TM-align, Many-core processors, Parallel algorithms, Master-slaves parallel computing.**

I. Introduction

The three-dimensional structure of a protein is known to have a strong correlation to its function [1]. Further, beyond evolutionary relationships encoded in the sequence, structure of proteins presents evidence of homology even in sequentially divergent proteins. Homology and functional similarity, due to their importance in applications such as drug design, have led pairwise protein structure comparison (PSC) to become an important research topic in computational biology over the last two decades. A newly discovered protein structure is typically compared with all known structures in order to ascertain its functional behavior. The faster this task can be performed the faster biologists and medical researchers can determine possible applications for the new protein. Improving the efficiency of protein structure comparison therefore is highly relevant to the field. The objective of the task is to retrieve a ranked list of proteins, where structurally similar proteins are ranked higher.

A typical task in bioinformatics is comparison of the structure of a protein with a database of known protein structures, one-to-many PSC, or when a set of multiple proteins is of interest, comparison of their structures to a whole database of known protein structures, many-to-many PSC. The unit operation in both these forms of PSC is the comparison of structures of a pair of proteins. Computational demand of the one-to-many and many-to-many PSC problem are a result of factors such as: exponential growth of structural proteomics databases, pairwise protein structure comparison is computationally intensive and several pairwise comparison approaches are typically of interest to the researcher.

Distributed computing solutions lend themselves to addressing some of these issues [2]. Using distributed setups computation heavy tasks can be broken down into smaller units, to be processed independently on different nodes, before combining the results to report to the user [2]. However, approaches leveraging distributed computing architectures do not address the problem of parallelizing pairwise structure comparison. Most PSC algorithms used in the domain continue to be serial and this highlights the possibility of further speed-ups by exploiting processor level parallelism. Modern many-core processor architectures can prove to be useful for exploiting available parallelism at this algorithmic level.

Several methods are used in the PSC domain and the current trend is to generate consensus results by combining them. The level at which parallel processing can be used in different methods depends on the underlying algorithms, making it important for the hardware platform employed to be flexible in terms of the resources allocated to the problem. Many-core processors can provide this flexibility, while retaining key elements of standard programming models, thus becoming of interest for the PSC domain. Furthermore, many-core processors provide a means for small research

978-0-7695-4979-8/13 $26.00 © 2013 IEEE
DOI 10.1109/IPDPSW.2013.222

setups to accelerate everyday tasks without depending on access to large-scale and expensive distributed computing infrastructures.

Algorithmic skeletons [3] allow a programmer to develop algorithms without specifying architecture dependent details. By nesting and combining skeletons desired level of parallelism can be introduced into different PSC methods. Using algorithmic skeletons allows programs to fully exploit the parallelism afforded by many-core processor architectures, while retaining the flexibility needed for experimenting with the level of parallelism and shared tasks (such as rotation of structures). While several algorithmic skeleton libraries have been developed over the last two decades [3], [4], few provide support for features such as type safety and to the best of our knowledge none has been targeted specifically for many-core processor architectures.

In this work we propose a framework for porting PSC algorithms to the Single-Chip Cloud Computer (SCC) experimental processor [5], which is a 48-core "concept vehicle" created by Intel Labs as a platform for many-core software research. We used the framework to port the popularly used TM-align algorithm [6] for the SCC and found that significant speedup can be achieved by efficiently utilizing the parallelism afforded by the processor. To the best of our knowledge this is the first implementation of PSC algorithms for many core processor architectures. The scalability of the many-core processors, in terms of cores, makes the speedup obtained by our method competitive with GPU based parallel implementations of TM-align [7]. Moreover, the method employed for porting TM-align can easily be applied to other PSC algorithms and extended to running multiple PSC algorithms in the same SCC processor chip.

The rest of the paper is organized as follows: In Section II we present the state-of-art of the PSC domain with emphasis on the modern trends. We develop a view of the source of complexity in the domain which results in the increased computational demand. We also briefly discuss the advent of Network-on-Chips (NoCs) and their application in scientific research. An overview of the Intel SCC from hardware and software perspectives is provided in Section III. In Section IV we present the framework for porting PSC algorithms and its implementation in porting TM-align for the SCC. Experimental results are presented and discussed in Section V.

II. Background and Related Work

Several methods have been developed to address the problem of protein structure comparison. Methods used vary greatly not only in the algorithmic techniques employed but also the similarity metrics used. A good overview of PSC methods, algorithms and metrics, can be found in [8], [9]. Of specific interest in this work is the TM-align [6] PSC method. The algorithm compares structures of proteins using TM-score rotation matrix and dynamic programming. Three kinds of initial alignments are used: dynamic programming based Secondary Structure Element alignment, gap less structure matching and dynamic programming alignment using scoring matrices obtained in the previous two alignments. A heuristic iterative algorithm is applied to the initial alignments in order to obtain the final one.

Due to the growing size of protein structure databases [10] and the need for building consensus methods, Multi-Criteria PSC (MC-PSC), the computational need in protein structure comparison has rapidly increased. This computational need has been met by developing faster heuristic algorithms and leveraging modern distributed architectures such as clusters and grids [2]. A pseudo-code representation of the one-to-all protein structure comparison problem using multiple comparison methods is shown in Algorithm 1.

Data:
q: query protein structure,
M: all protein structure comparison methods,
D: database of known protein structures
for *k in M* **do**
  for *i in D* **do**
    compute(in n, using method k, protein pair *[i, q])*
  end
end

Algorithm 1: A pseudo-code (adapted from [2]) for multi-method protein structure comparison. Each node n performs a pairwise comparison of proteins (i,q) using one of the methods (k) belonging to all methods of interest M. The method essentially finds a free compute node and passes the 'job' to the node. The approach expands to any number of computational nodes available.

The predominant approach, which is making grid computing available for PSC and MC-PSC, is showcased in [11]–[18]. Application of processor technologies such as Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) to the broader field of structural proteomics can be found in [7], [19], [20].

Multi-core processor architectures have also grown in availability with dual and quad core processors becoming mainstream in desktops and servers [21]. A key advantage of many-core architectures is the availability of familiar programming models, languages and tools. Initiatives such as Many Integrated Core Architectures (MIC) from Intel, make CPU architectures with super computing capability packaged on a single chip available for scientific computing purposes.

With multiple cores appearing on a single chip, fast inter-core communication becomes important. Two broad strategies, through which inter-core communication is performed, are: (a) using a single communication bus, and (b) using an interconnection network. Due to the scope for supporting

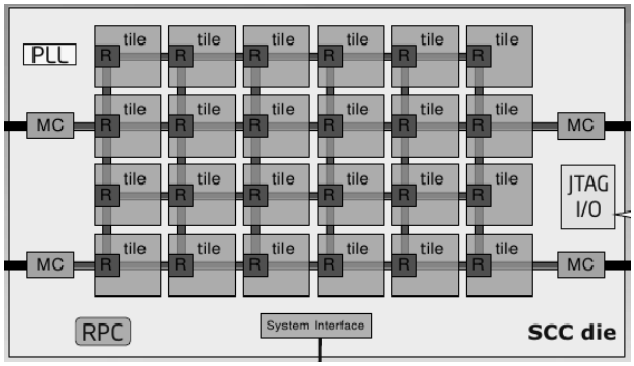

Figure 1. Intel SCC System Overview [5]. A view of the SCC chip showing routers (R), memory controllers (MC) and 24 tiles.

large number of cores the use of networks for inter-core communication is gaining in popularity [22]. The resulting many-core processors use Networks on Chip (NoC) [23] as an approach for the communication subsystem.

Attempts have been made to utilize many-core architectures in bioinformatics applications [24]. The majority of the applications attempted belong to the pairwise or multiple sequence comparison category. The NoC based implementation of the sequence alignment algorithm by Needleman and Wunsch [25] developed in [26], shows significant speedup potential. Despite the flexibility and parallel processing capabilities, NoCs have not yet been extensively exploited in bioinformatics [26] and to the best of our knowledge no protein structure comparison algorithms have been ported for a many-cores processor.

III. Intel Single-Chip Cloud Computer

The "Single-chip Cloud Computer" (SCC) [27], is an experimental Intel architecture microprocessor containing 48 cores integrated on a silicon CPU chip [27], [28]. It has multiple dual x86 core tiles arranged in a 6x4 grid, memory controllers and 24-router mesh network, depicted in Figure 1. The technology used is scalable to support more than 100 cores on a single chip [29]. The cores on the chip can run separate operating systems acting like independent computational nodes that communicate with other nodes over a packet-based network.

A. Hardware Architecture

The SCC resembles a cluster of computer nodes capable of communicating with each other in the same way as a cluster of independent machines. Salient features of the SCC hardware architecture relevant to programming the chip are listed in Table I.

Figure 2, shows details of individual tiles on the SCC many-core processor. Each core of the SCC has L1 and L2 caches. In the SCC architecture, the L1 caches (16KB each) are on the core while the L2 caches (256KB each)

Table I
Salient features of the SCC Chip by Intel.

Core architecture	6x4 mesh, 2 Pentium P54c (x86) cores per tile
Local cache	256KB L2 Cache, 16KB shared MPB per tile
Main memory	4 iMCs, 16-64 GB total memory

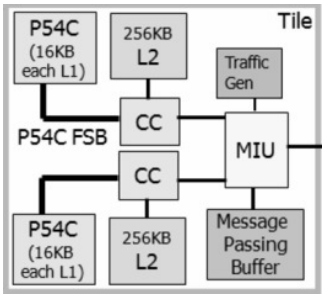

Figure 2. Each of the 24 tiles in the SCC processor contains 2 cores with L1 and L2 caches and a message passing buffer (MPB). The SCC layout and tile architecture showing routers (R), memory controllers (MC), mesh interface units (MIU), cache controllers (CC), and second generation Pentium processor cores (P54C) with their front side bus (FSB) [30].

are on the tile next to the core with each tile carrying 2 cores. Further, each tile also has a small message passing buffer (MPB), of 16KB, and is shared among all the cores on the chip. Hence, with 24 tiles, the SCC provides a message passing buffer of size 384KB. The SCC therefore provides a hierarchy of memories usable by application programs for different purposes including processing and communication. The Mesh Interface Unit (MIU), connecting the tiles to the mesh, builds packets from data to put it in the mesh and unpacks data coming in from the mesh.

B. Software Architecture

From a programmers perspective the SCC provides a super computing environment where the paradigm shift from standard programming is small. Programming a standard x86 core and using Message Passing Interface for fast message exchange between processes is currently employed widely for cluster programming [30]. The cores of the SCC are visible to host machine to which the SCC is attached, Management Console PC (MCPC), with names of the form rck00, rck01..., rck47. In order to facilitate development, a minimal programming library RCCE [31], a compact, lightweight communication environment written in C, is available with a basic API for MPI. The main features of the RCCE library are:

- RCCE supports message passing APIs and allows mapping tasks onto many-core chips.
- RCCE usage requires parallel programming experience with an understanding of the underlying SCC hardware.

With the help of RCCE it is possible to build full scale application programs for the SCC chip. The simple message passing environment provided by the RCCE makes it easy

to build systems with one-sided communication, such as is sufficient for parallelizing algorithms. Software for the SCC can be compiled using the Intel C/C++ compiler or the GNU C/C++ compiler (gcc). Since the cores on the SCC are x86 type the compilation must be performed for a 32 bit system.

A program written using RCCE is typically expected to belong to the Single program multiple data (SPMD) computing model. Detailed analysis of the memory address space resolution can be found in [30]. From the programmers view each instance of the program runs on a different core of the SCC. Message passing is employed, using the RCCE, in order to synchronize the processes. RCCE also provides the programmer with constructs for creating a shared memory space that multiple cores can access.

To the best of our knowledge the parallel processing capabilities of the SCC have not been leveraged for computational proteomics applications. The work presented here is the first attempt at porting PSC algorithms to the architecture.

IV. Framework for porting Protein Structure Comparison algorithms for many-core processors

In this section we present the framework we used for porting a popular PSC algorithm, TM-align, to many-cores processor technology. The approach can be extended to MC-PSC as discussed later in Section V. Typically PSC involves one-to-all or all-to-all comparisons of protein structures. Each pairwise structure comparison is an independent unit operation. Several pairwise comparison operations can therefore be performed in parallel if the computing resources are available. Many-core processors provide several computing elements, connected by a high speed network allowing distribution of jobs.

We propose the use of a master-slaves parallel implementation of the PSC algorithm, where the master process is responsible for loading the structures to be compared and distributing the pairwise comparison jobs to the slave processes. In a many-core processor system, where the cores are connected by a high speed interconnection network, the data transfer overhead is relatively small. By limiting data loading to a single process we avoid bottlenecks, created due to multiple processes accessing the same data concurrently. The slave processes perform pairwise structure comparison on structure data received from the master and return results of processing to the master. The slave processes continue the cycle until they receive a terminate signal from the master. This simple strategy, where the master process polls the slave processes in a round-robin manner, allows efficient use of the computing resources available to perform pairwise protein structure comparisons.

The rckskel library

In order to facilitate development of PSC algorithms targeted for the SCC, we built a C library. The library provides convenient wrappers for common operations, such as environment initialization, testing how many cores are available to the program, setting debug levels and finalization, performed by all applications built with the RCCE. Further, we found that higher level constructs which hide the details of the inter-process communication, e.g. polling and waiting, would simplify introducing parallelism in PSC algorithms. To retain the flexibility offered by RCCE, in combining processes running on different cores to form a pipeline or to perform parallel execution, we decided to use algorithmic skeletons [3] for building the library.

The library we developed, called *rckskel* (rck Skeleton Library), is a small parallel programming library which implements algorithmic skeletons in addition to providing convenient wrappers for common RCCE related tasks. A list of skeleton functions implemented in *rckskel* follows:

- SEQ: This is a task sequencing construct where a list of tasks, which may contain sub-tasks, are assigned to a set of processing elements but will be executed only in the order in which they were given. This construct is typically useful for defining the leaf node operations in a hierarchy of operations. Parameters to the function include, *ue_count* (the number of processing elements), *ue_ids* (the specific processing elements), *check_ready* (function to be used for checking if a processing element has been initialized) and *task_count* (the number of sub-tasks). The dots at the end of the function definition denote a variable argument list of tasks. This construct runs the jobs on the corresponding processing elements sequentially. Once the last batch of jobs is submitted it returns to the calling code, without waiting for them to complete.

```
void SEQ(int ue_count, int *ue_ids,
  int (*check_ready)(int),
  int task_count,...);
```

- PAR: This is a task mapping construct where each task, which may contain sub-tasks, is assigned a set of processing elements and the sub-tasks are processed in parallel. This construct assigns the jobs to the corresponding processing elements and returns to the calling code, without blocking till the jobs are completed. The calling code will need to call the COLLECT construct if it needs to wait for the jobs to finish.

```
void PAR(int ue_count, int *ue_ids,
  int (*check_ready)(int),
  int task_count, ...)
```

- COLLECT: This is a task collection construct where a list of processing elements is polled, till all elements

return the results of their processing. The function to be applied to the data returned can be specified and may perform operations on the returned data, e.g. storing in an array for later use. The function does not block waiting on processing elements in order but rather performs a busy round-robin loop.

```
void COLLECT(int ue_count,
  int *ue_ids, int (*collector)(int));
```

- FARM: This is a master-slaves task execution construct. A controlling task is created which ensures the execution of the tasks given until they complete. The master process in this setup runs on one of the cores of the SCC. This is the highest level construct currently implemented and takes care of ensuring that all processing elements are available before starting the processing and that all processing is completed when it returns. A task tree is generated from the parameters of the function depending on the sub-tasks. If no tasks have been specified FARM works as a PAR followed by a COLLECT. The tasks in the tree are processed as specified, in parallel or in sequence, using the PAR, SEQ and COLLECT constructs described above.

```
void FARM(int ue_count, int *ue_ids,
  int (*check_ready)(int),
  int task_count, ...);
```

A template for a master-slaves implementation utilizing *rckskel* is shown in Figure 3. The function name *RCCE_APP* is the entry point for applications built with RCCE. The division of master and slave related code, after the common variable declaration and environment initializations, is highlighted by the *if* block, as typical for SPMD-style code. The *MASTER_ID* is defined globally with a default value of 0, which implies that the first core available to the program will be used to run the master process. The master processing starts with creation of a FARM task where application specific methods, *master_send_job* and *master_receive_result*, are supplied. These are application specific because the data structure used by different applications vary.

The jobs to be processed, as well as the SCC cores on which the jobs should be run (*ue_ids*) are also specified in the task definition. The jobs are run using the *FARM* execution construct. The *check_ready* method supplied is used to start distribution of a job to a slave when the slave becomes ready. The slave processing proceeds by waiting in a busy loop, receiving data from the master process and returning results once the processing is complete, until it receives a terminate message.

The *client_receive_job* method contains a blocking wait on the master and application specific processing of data. The results are returned to the master in an application specific data structure of type *RESULT_BLOCK*.

```
int RCCE_APP(int argc, char **argv){
  // variable declaration & environment setup
  ...
  task_t *task;
  if(ue_id == MASTER_ID) {
    task = create_task(MAP_rckskel, '0',
      &master_send_job, &master_receive_result,
      job_indexes, ue_ids);
    FARM(ue_count, ue_ids, &check_ready,
      1, task);
  } else {
    // local intializations
    ...
    while(TRUE_) {
      // get and process job or terminate
      if(client_receive_job() == TRUE_) break;
      RCCE_send((char *)&result,
        sizeof(RESULT_BLOCK), MASTER_ID);
    }
  }
  ...
}
```

Figure 3. Template C code for setting up master-slaves processing using *rckskel*. The user defined function $client\_receive\_job$, blocks waiting for a job from the master. On receiving a job, it executes a user defined function for ex. a PSC method.

The *rckskel* library is implemented on top of RCCE making it directly usable for building software applications that can be run on the SCC. It provides easy to use data structures for wrapping tasks and defining the SCC cores (processing elements) to be used. Message passing between processes running on different cores is performed using the *RCCE_send* and *RCCE_recv* functions, hence no shared memory allocation operation is performed. In its current form, the master process performs a round-robin polling of children processes it controls.

From a programmer's perspective, it is important to clarify the difference between 'task' and 'job' as used in *rckskel*. A job refers to an application specific data structure describing the processing to be performed e.g. a pairwise PSC is a job and would specify the structures of the proteins to be compared and the method called for comparing them. A task refers to a collection of jobs, or other tasks e.g. one-vs-all PSC is a task. Similarly, multiple one-vs-all PSC tasks (many-vs-all PSC) would also constitute a task, forming a hierarchy of tasks. Thus the task data structure is used to capture jobs to be processed, the manner in which they must be processed (serial or parallel) and the computing resources available (SCC cores) to them. Allocation of a core to a job is performed dynamically, but the cores available are restricted to those supplied to the task. Thus allocating a sensible number of cores, based on the number of jobs, is left to the software implementation.

```
int client_receive_job() {
  // wait to receive instruction from master
  RCCE_recv((char *)&client_in_data,
    sizeof(MasterToClientTransferBlock), MASTER_ID);
  // early exit if master is asking to leave
  if(client_in_data.die == TRUE_) {
    return TRUE_;
  }
  // do TMalign
  ... local assignments
  tmalign(&coords1, &coords2, &seq1, &seq2);
  // store results in appropriate structures
  return FALSE_;
}
```

Figure 4. Outline C code showing implementation of the $client\_receive\_job$ job function used in $rckalign$. The function is used as shown in Figure 3

The rckAlign application

We developed a port of the TM-align algorithm, called *rckAlign*, for Intel's many-core SCC processor using the *rckskel* library. The parallel algorithm developed makes use of the master-slaves implementation provided by the library. The implementation contains a single master process which generates a list of jobs, each involving a single pairwise protein structure comparison. The jobs are then processed in parallel on all the slave processes available to the master. The first core supplied to the program is used to run the master process and all subsequent cores are used to run slave processes.

In order to develop the parallel implementation, we first ported TM-align software to a pure C implementation. The Fortran code of TM-align was converted to C using the F2C converter [32]. The resulting code had a dependency on the F2C library and was therefore cleaned up manually to remove all dependencies. This required altering the data types, as well as implementing four basic math functions: *max*, *min*, *dabs* and *abs*. Additionally, the I/O operations were changed to use C functions. We compared the output of the C implementation with that of the Fortran implementation and found matching results. The C implementation of TM-align was used to generate the baseline results used in the performance comparisons reported in Section V. A master-slaves parallel implementation of the C code, using the *rckskel* library and the template described in Figure 3 was then generated.

As evidenced by Figures 3 and 4, using the *rckSkel* library a typical PSC method can easily be parallelized, enabling it to exploit the parallelism offered by the SCC. The *rckSkel* FARM function together with the user written functions shown in these figures, allow the main PSC algorithm to be implemented serially (in the function $tmalign$) and ensure maximum resource utilization. No further code-complexity is introduced regardless of the number of SCC cores used by the software at run time.

V. Results and Discussion

In this section, we describe the experiments performed, to validate the performance characteristics of *rckAlign*, present the experimental results and discuss them. Comparisons of the parallel and serial implementations of the TM-align PSC method were performed by running all-vs-all tasks and recording the execution times. The speedup achieved, as compared to the existing serial software, was measured as a function of the number of cores used in the many-cores processor architecture.

A. Experimental systems

We benchmarked *rckAlign* by performing all-vs-all PSC experiments using the SCC. The first baseline for benchmarking was obtained by running the same all-vs-all PSC experiment with the existing TM-align software using a 2.4 GHz AMD CPU with 3 GB RAM, running Debian Stable. The second baseline was running the algorithm on a single core of the SCC i.e. an P54C Intel Pentium core at 800 MHz, running SCC Linux. It must be noted that currently available TM-align software and the C port used in our experiments are serial implementations and do not take advantage of multi-core systems. All software, both for the PC and the SCC, was compiled using the GNU C Compiler version 4.7.

B. Datasets

The all-vs-all PSC experiments were performed for two protein domain datasets. The datasets were generated by using the first chain of the first model for the Rost and Sander dataset (RS119) [33] and for the Chew-Kedem dataset (CK34) [34]. While the Rost and Sander dataset contains 119 protein structure chains, the Chew-Kedem dataset contains only 34. These datasets have previously been used in performance comparison of PSC methods [2], [35].

C. Experiment I

We compared *rckAlign* running on the SCC and the existing TM-align software used in a distributed manner, to perform all-vs-all comparison for the CK34 dataset. In the distributed TM-align version, a controlling master process is run on the SCC host machine (MCPC). The host process creates a list of jobs and distributes them to individual cores of the SCC. Each process is responsible for loading its own structure data. Issuing a job to a core is performed using the *pssh* remote execution command available on the MCPC. On the other hand, when using *rckAlign* the data is loaded by the master process and slave processes receive the data for pairwise structure comparison from the master process using the SCC network. Results of the experiment are shown Figure 5 with detailed values presented in Table II.

As observed in Figure 5 and Table II, *rckAlign* achieves faster processing times than when the master process is running on the MCPC. There are two main reasons for

Table II
COMPARISON OF PARALLEL *rckAlign* TO DISTRIBUTED TM-ALIGN IN AN ALL-VS-ALL PSC TASK USING THE CK34 DATASET. ALL TIMES ARE IN SECONDS.

Slave Cores	Algorithm	
	rckAlign	**TM-align**
1	2027	5212
3	689	1704
5	420	854
7	305	569
9	238	511
11	196	452
13	168	382
15	148	332
17	132	293
19	120	262
21	109	238
23	101	218
25	94	202
27	88	187
29	83	175
31	79	168
33	73	174
35	71	173
37	68	145
39	65	143
41	62	132
43	60	126
45	59	122
47	56	120

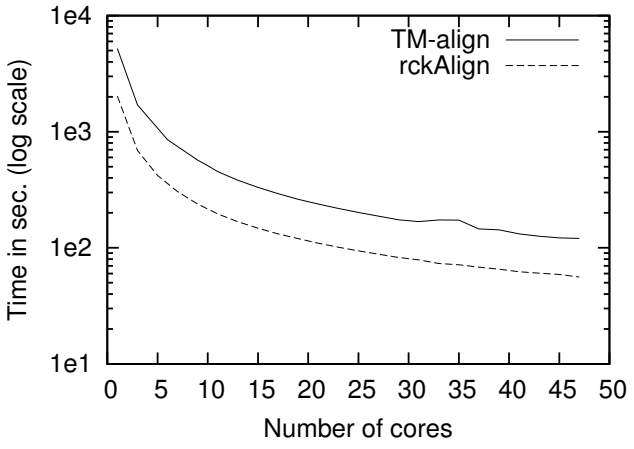

Figure 5. Performance comparison of parallel *rckAlign* with that of distributed TM-align software (C port) for the Chew-Kedem dataset (CK34) as the number of slave cores used is increasing.

this behavior: (a) disk access through the Network File System (NFS) creates a bottleneck when multiple processes are trying to access the data, and (b) high environment setup costs incurred while issuing remote processing. The SCC-MCPC setup provides NFS access to disks installed in the MCPC for the Linux system running on each individual core. When processes running on several cores try to access the data stored in the shared partition of the MCPC disk a bottleneck is created, by the MCPC disk controller, resulting in overall increase in processing time. This situation is not encountered when all the protein related data is loaded by a single process as is the case for *rckAlign*. Further, the master process running on the MCPC starts a new process for each pairwise comparison, which has its environment setup cost, thus increasing further the total processing time. Results of the comparison thus validate the superiority of the approach where the master process runs on one of the SCC cores rather than on the host PC machine. Additional overhead cost is also avoided in *rckAlign* because all processes, master and slaves, are initialized once for a given number of slaves.

Table III
TIME REQUIRED FOR THE BASELINE ALL-VS-ALL PSC TASK USING TM-ALIGN (C PORT) ON TWO DIFFERENT PROCESSORS AND DATASETS. ALL TIMES ARE IN SECONDS.

Processor	Datasets	
	CK34	**RS119**
AMD Athlon II X2 250 2.4 GHz	406	7298
Intel P54C Pentium 800 MHz	2029	28597

D. Experiment II

The times required for performing the all-vs-all comparisons for the two datasets using the serial implementation of TM-align were measured, both for the AMD Athlon II X2 250 2.4 GHz processor and for the P54C Intel Pentium SCC Core at 800 MHz. The times obtained for the SCC core were used as the baseline for calculating the speedup achieved by *rckAlign* running in parallel on the SCC. For running on a single SCC core, the TM-align program was modified slightly, to load all the protein structures to be compared at the start in order to be equivalent to the way *rckAlign* works. Results of the experiment are presented in Table III. When comparing baselines performance (using one core) the faster AMD CPU outperforms as expected the much slower Intel P54C core.

The speedup achieved by the parallel *rckAlign* implementation on the SCC was measured for both datasets CK34 and RS119. This experiment was designed to measure the speedup achieved as a function of the number of slave processes used as well as the size of the datasets. The master process loads all the domains to be processed and creates a list of jobs with all pairs (all-vs-all). The master process then distributes N jobs among the N slaves and the results

Table IV
PERFORMANCE OF RCKALIGN IN AN ALL-VS-ALL PSC TASK ON THE CK34 AND RS119 DATASETS.

Slave Cores	CK34		RS119	
	Speedup	Time (sec)	Speedup	Time (sec)
1	1	2029	1	28597
3	2.94	689	2.96	9654
5	4.82	420	4.91	5818
7	6.66	305	6.95	4114
9	8.52	238	8.94	3195
11	10.34	196	10.97	2605
13	12.09	168	12.95	2208
15	13.74	148	14.88	1921
17	15.36	132	16.76	1705
19	16.89	120	18.64	1534
21	18.53	109	20.59	1389
23	20.03	101	22.52	1270
25	21.56	94	24.52	1166
27	23.02	88	26.49	1079
29	24.52	83	28.45	1005
31	25.72	79	30.37	941
33	27.68	73	32.32	885
35	28.43	71	34.21	836
37	29.75	68	36.14	791
39	30.97	65	38.01	752
41	32.60	62	39.74	719
43	33.59	60	41.49	689
45	34.45	59	43.40	659
47	36.17	56	44.78	640

are gathered by polling the slaves in a round-robin manner. The distribution of jobs and collection of results is carried out until all jobs have finished. Communication between the master and the slaves is carried out using functions available in the *rckskel* API. The number of active slaves was varied from 1 to 47 in order to assess the impact of increasing the number of cores available for parallel processing. Results of the experiment are shown in Figure 6 with detailed values presented in Table IV.

Figure 6 shows that the speedup achieved by *rckAlign* is increasing almost linearly with the number of cores available for running slave processes. This is a result of the low cost of exchanging data between processes running on cores connected by a high speed interconnection network. If the data transfer times were high, the master process would become a bottleneck and core utilization would be reduced, resulting in larger overall processing times. Since an almost linear speedup is observed, the simple master-slaves implementation appears sufficient to exploit the parallelism offered by many-core processors to this problem. This observation suggests that further speedup can be achieved

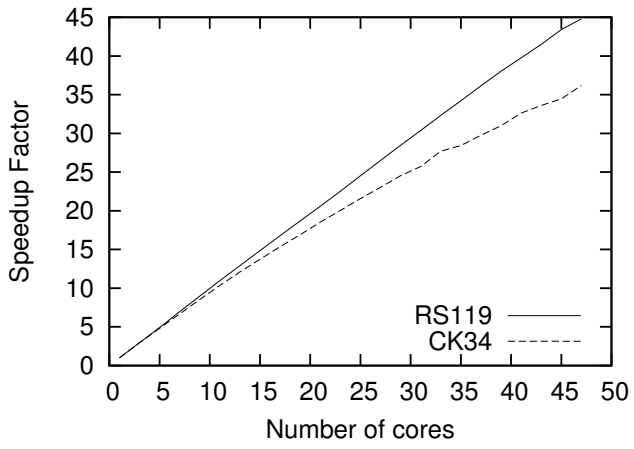

Figure 6. Speedup achieved by *rckAlign* as the number of slave cores is increasing (from 1 to 47) for the Chew-Kedem (CK34) and the Rost-Sanders (RS119) datasets. The speedup reported is relative to the performance on a single core of the SCC.

Table V
COMPARISON OF TIMES REQUIRED BY TM-ALIGN AND *rckAlign* FOR PERFORMING ALL-VS-ALL PSC ON THE CK34 AND RS119 DATASETS. ALL TIMES ARE IN SECONDS.

Dataset	TM-align AMD@2.4GHz	TM-align Intel@800MHz	rckSkel SCC(all cores)
CK34	406	2029	56
RS119	7298	28597	640

on many-core processors with a greater number of cores. It should be mentioned that no load balancing was applied to the allocation of jobs to slaves in our implementation. It has been suggested that good load balancing approaches can improve the performance of all-vs-all PSC [2]. We are currently investigating if such techniques could further improve the performance of *rckAlign*.

From the results in Table V we observed that *rckAlign* running in the SCC at 800 MHz achieves an 11 fold speedup over the AMD 2.4GHz processor and a 44 fold speedup over a single Intel P54C 800 MHz processor when using the RS119 dataset. We also observe that the larger the dataset the higher the speedup observed. These results suggest that many-core NoCs with fast interconnection networks and faster processor cores than the SCC will be ideal candidates for delivering high performance for all-to-all PSC tasks applied to large size protein databases, as needed for combinatorial drug design.

Comparison of the performance of *rckAlign* with the serial implementations run on a single core of the SCC and the use of the faster processor, suggest that there is scope for achieving higher overall speedups, if the many-core processor provides faster cores. It is possible that the single master strategy would become the bottleneck, if slave processes were running on faster cores or faster network.

However, this can be tackled by implementing a hierarchy of master processes such that a master does not become a bottleneck for the slaves it controls.

Finally, the approach developed in this work can be extended to the more general MC-PSC problem, since all slave processes are not required to run the same PSC algorithm. The basic protein structure data used by most PSC algorithms is the same and therefore, different slave processes can be running different algorithms on the same data received from the master process (in algorithm specific data structures). Such an extension of the approach would require assessment of optimal strategies for the partitioning of the cores dedicated to different PSC algorithms, since the algorithm complexities may vary. However, since the partition of cores to different tasks is implementation specific, from a software development standpoint this can be facilitated using the library developed for this work.

VI. Conclusions

Computational challenges in the protein structure comparison (PSC) problem are a result of several factors: structural proteomics databases getting larger at a very fast pace, high computational complexity of pairwise protein comparison algorithms and the multitude of pairwise comparison approaches popularly used in the field. Advances in modern processor architectures, such as many-core CPUs, have enabled them to support parallelism making them a natural candidate for speeding up all-to-all PSC, an essential operation in computational proteomics for drug design. We have introduced a framework using algorithmic skeletons for porting algorithms to a many-core processor architecture. Using this framework, we have also developed a parallel implementation, called *rckalign*, for Intel's SCC many-core processor of the popular TM-align PSC algorithm. Experimental results, comparing the performance of *rckAlign* with the existing serial implementation, demonstrate that almost linear speedup can be achieved by leveraging the parallelism offered by many-core processors and their fast interconnection networks.

Given the trends in the domain, future work will involve extending the framework to support all-to-all multi-criteria PSC and studying the performance characteristics of such a system. Furthermore, building support for threading into the base library will be investigated, since this can be critical when the protein structure datasets are too large to be loaded into memory at once.

Acknowledgments

Mr. Sharma and Dr. Manolakos would like to acknowledge the support of this research by the European Union (European Social Fund ESF) and Greek national funds, through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program "Heracleitus II". We would also like to acknowledge the MicroLab - ECE - National Technical University of Athens, Greece for allowing use of their SCC infrastructure, performed under the supervision of Dr. Antonis Papanikolaou. In addition we thank Dimitrios Rodopoulos and Prof. Dimitrios Soudris of NTUA for providing comments and assistance.

References

[1] R. A. Laskowski, J. D. Watson, and J. M. Thornton, "ProFunc: a server for predicting protein function from 3D structure," *Nucleic Acids Research*, vol. 33, pp. 89–93, 2005.

[2] A. A. Shah, G. Folino, and N. Krasnogor, "Toward high-throughput, multicriteria protein-structure comparison and analysis," *IEEE Transactions on NanoBioscience*, vol. 9, no. 2, pp. 144–155, 2010.

[3] H. González-Vélez and M. Leyton, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Softw. Pract. Exper.*, vol. 40, no. 12, pp. 1135–1160, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1002/spe.v40:12

[4] D. K. G. Campbell, "Towards the Classification of Algorithmic Skeletons," Tech. Rep. YCS 276, 1996. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.8909

[5] The SCC Platform Overview. [Online]. Available: http://techresearch.intel.com/spaw2/uploads/files/SCC_Platform_Overview.pdf

[6] Y. Zhang and J. Skolnick, "TM-align: a protein structure alignment algorithm based on the TM-score," *Nucleic Acids Research*, vol. 33, no. 7, pp. 2302–2309, 2005. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/15849316

[7] B. Pang, N. Zhao, M. Becchi, D. Korkin, and C.-R. Shyu, "Accelerating large-scale protein structure alignments with graphics processing units." *BMC Res Notes*, vol. 5, no. 1, p. 116, 2012. [Online]. Available: http://www.biomedsearch.com/nih/Accelerating-large-scale-protein-structure/22357132.html

[8] G. Lancia and S. Istrail, "Protein structure comparison: Algorithms and applications," in *Mathematical Methods for Protein Structure Analysis and Design*, 2003, pp. 1–33.

[9] I. Eidhammer, I. Jonassen, and W. R. Taylor, "Structure comparison and structure patterns," *Journal of Computational Biology*, vol. 7, no. 5, pp. 685–716, 2000.

[10] P. Chi, "Efficient protein tertiary structure retrievals and classifications using content based comparison algorithms," Ph.D. dissertation, Columbia, MO, USA, 2007.

[11] D. Barthel, J. Hirst, J. Blacewicz, and N.Krasnogor, "ProCKSi: a Metaserver for Protein Comparison Using Kolmogorov and Other Similarity Measures," *BMC Bioinformatics*, vol. 8, p. 416, 2007.

[12] A. Krishnan, "A survey of life sciences applications on the Grid," *New Generation Computing*, vol. 22, no. 2, pp. 111–125, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1007/BF03040950

[13] M. Cannataro, C. Comito, F. L. Schiavo, and P. Veltri, "Proteus, a grid based problem solving environment for bioinformatics: Architecture and experiments," *IEEE Computational Intelligence Bulletin 3*, vol. 1, pp. 7–17, 2004.

[14] N. Krasnogor, A. A. Shar, D. Barthel, P. Lukasiak, and J. Blazewicz, "Web and grid technologies in bioinformatics, computational and systems biology: A review," *Current Bioinformatics*, vol. 3, no. 1, pp. 10–31, 2008. [Online].

Available: http://www.ingentaconnect.com/content/ben/cbio/2008/00000003/00000001/art00002

[15] C. Ferrari, C. Guerra, and G. Zanotti, "A grid-aware approach to protein structure comparison," *Journal of Parallel and Distributed Computing*, vol. 63, no. 7-8, pp. 728–737, 2003. [Online]. Available: http://portal.acm.org/citation.cfm?id=952910

[16] A. A. Shah, D. Barthel, and N. Krasnogor, "Grid and distributed public computing schemes for structural proteomics : A short overview," *Frontiers of High Performance Computing and Networking ISPA 2007 Workshops*, vol. 4743, pp. 424–434, 2007. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74767-3_44

[17] M. Cannataro, M. Comin, C. Ferrari, C. Guerra, A. Guzzo, and P. Veltri, "Modelling a protein structure comparison application on the grid using PROTEUS," in *Proceedings of the First international conference on Scientific Applications of Grid Computing*, ser. SAG'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 75–85. [Online]. Available: http://dx.doi.org/10.1007/11423287_7

[18] D. Pekurovsky, I. N. Shindyalov, and P. E. Bourne, "A case study of high-throughput biological data processing on parallel platforms," *Bioinformatics*, vol. 20, no. 12, pp. 1940–1947, 2004. [Online]. Available: http://www.ncbi.nlm.nih.gov/pubmed/15044237

[19] K. Karplus, R. Karchin, J. Draper, J. Casper, Y. Mandel-Gutfreund, M. Diekhans, and R. Hughey, "Combining local-structure, fold-recognition, and new fold methods for protein structure prediction," *Proteins*, vol. 53, pp. 491–496, 2003. [Online]. Available: http://dx.doi.org/10.1002/prot.10540

[20] A. D. Stivala, P. J. Stuckey, and A. I. Wirth, "Fast and accurate protein substructure searching with simulated annealing and GPUs," *BMC Bioinformatics*, vol. 11, no. 1, p. 446, 2010. [Online]. Available: http://www.biomedcentral.com/1471-2105/11/446

[21] M. Azimi, N. Cherukuri, D. Jayashima, A. Kumar, P. Kundu, S. Park, I. Schoinas, and A. Vaidya, "Integration Challenges and Tradeoffs for Tera-scale Architectures," *Intel Technology Journal*, vol. 11, pp. 173–184, 2007.

[22] P. Guerrier and A. Greiner, "A generic architecture for on-chip packet-switched interconnections," in *Proceedings of the conference on Design, automation and test in Europe*, ser. DATE '00. New York, NY, USA: ACM, 2000, pp. 250–256. [Online]. Available: http://doi.acm.org/10.1145/343647.343776

[23] D. Atienza, F. Angiolini, S. Murali, A. Pullini, L. Benini, and G. De Micheli, "Network-On-Chip Design and Synthesis Outlook," *INTEGRATION*, vol. vol. 41, n. 2, pp. 1 – 35, 2008, [ARTICOLO].

[24] S. Isaza, "Multicore architectures for bioinformatics applications," Ph.D. dissertation, October 2011.

[25] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins." *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, Mar. 1970. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/5420325

[26] S. Sarkar, G. R. Kulkarni, P. P. Pande, and A. Kalyanaraman, "Network-on-Chip Hardware Accelerators for Biological Sequence Alignment," *IEEE Transaction on Compututation*, vol. 59, no. 1, pp. 29–41, Jan. 2010. [Online]. Available: http://dx.doi.org/10.1109/TC.2009.133

[27] "Single-chip Cloud Computer," URL:http://techresearch.intel.com/ProjectDetails.aspx?Id=1. [Online]. Available: http://techresearch.intel.com/ProjectDetails.aspx?Id=1

[28] "Single-chip Cloud Computer," http://www.techopedia.com/definition/26749/single-chip-cloud-computer. [Online]. Available: http://www.techopedia.com/definition/26749/single-chip-cloud-computer

[29] B. Marker, E. Chan, J. Poulson, R. van de Geijn, R. F. Van der Wijngaart, T. G. Mattson, and T. E. Kubaska, "Programming many-core architectures - a case study: dense matrix computations on the Intel single-chip cloud computer processor," *Concurrency Computat.: Pract. Exper.*, vol. 24, pp. 1317–1333.

[30] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core SCC Processor: the Programmer's View," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.53

[31] "RCCE: a small library for Many-Core communication," URL:http://techresearch.intel.com/spaw2/uploads/files//RCCE_Specification.pdf. [Online]. Available: http://techresearch.intel.com/spaw2/uploads/files//RCCE_Specification.pdf

[32] Fortran to C conversion library (F2C). [Online]. Available: http://www.netlib.org/f2c/

[33] B. Rost and S. C, "Prediction of protein secondary structure at better than 70% accuracy," *Journal of Molecular Biology*, vol. 253, pp. 584–599, 1993.

[34] L. P. Chew and K. Kedem, "Finding the consensus shape for a protein family (extended abstract)," in *18th ACM Symposium on Computational Geometry*, 2002, pp. 64–73.

[35] D. Barthel, J. D. Hirst, J. Blazewicz, E. K. Burke, and N. Krasnogor, "ProCKSI: a decision support system for protein (structure) comparison, knowledge, similarity and information," *BMC Bioinformatics*, vol. 8, p. 416, Oct. 2007. [Online]. Available: http://dx.doi.org/10.1186/1471-2105-8-416