Published in final edited form as:

Netw Model Anal Health Inform Bioinform. 2014 April ; 3: 54–. doi:10.1007/s13721-014-0054-1.

Exploiting Thread-Level and Instruction-Level Parallelism to Cluster Mass Spectrometry Data using Multicore Architectures

Fahad Saeed^{1,2,*}, Jason D. Hoffert², Trairak Pisitkun^{2,3}, and Mark A. Knepper²

¹Department of Computer Science and, Department of Electrical & Computer Engineering, Western Michigan University, Kalamazoo, MI USA ²Epithelial Systems Biology Laboratory, National Heart Lung and Blood Institute (NHLBI), National Institutes of Health (NIH), Bethesda, MD USA ³Faculty of Medicine, Chulalongkorn University, Bangkok, Thailand

Abstract

Modern mass spectrometers can produce large numbers of peptide spectra from complex biological samples in a short time. A substantial amount of redundancy is observed in these data sets from peptides that may get selected multiple times in Liquid Chromatography Tandem Mass Spectrometry (LC-MS/MS) experiments. A large number of spectra do not get mapped to specific peptide sequences due to low signal-to-noise (S/N) ratio of the spectra from these machines. Clustering is one way to mitigate the problems of these complex mass spectrometry data sets. Recently we presented a graph theoretic framework, known as CAMS, for clustering of large-scale mass spectrometry data. CAMS utilized a novel metric to exploit the spatial patterns in the mass spectrometry peaks which allowed highly accurate clustering results. However, comparison of each spectrum with every other spectrum makes the clustering problem computationally inefficient. In this paper we present a parallel algorithm, called P-CAMS, that uses thread-level and instruction-level parallelism on multicore architectures to substantially decrease running times. P-CAMS relies on intelligent matrix completion to reduce the number of comparisons, threads to run on each core and Single Instruction Multiple Data (SIMD) paradigm inside each thread to exploit massive parallelism on multicore architectures. A carefully crafted load-balanced scheme that uses spatial locations of the mass spectrometry peaks mapped to nearest level cache and core allows super-linear speedups. We study the scalability of the algorithm with a wide variety of mass spectrometry data and variation in architecture specific parameters. The results show that SIMD style data parallelism combined with thread-level parallelism for multicore architectures is a powerful combination that allows substantial reduction in runtimes *even* for all-to-all comparison algorithms. The quality assessment is performed using real-world data set and is shown to be consistent with the serial version of the same algorithm.

*Correspondence should be addressed to Fahad Saeed at fahad.saeed@nih.gov.

Notations of Fset and F-set are used interchangeably in this paper

1 Introduction

Mass Spectrometry analysis is an integral part of modern large-scale proteomics studies. Mass spectrometers can generate thousands of spectra in a single run and are useful in large-scale protein identification and quantitation studies [1] [2] [3]. A typical mass spectrometer works by ionizing molecules introduced at the ion source in the form of liquid solutions. These charged ions are then desolvated and transferred in the gas phase as ions that are then further processed to get hundreds of thousands of these complex stochastic spectra[4]. Protein mass Spectrometry has proven very useful for basic and clinical biological research [1, 5, 6].

Protein mass spectrometers developed over the past few years have grown incredibly efficient and can generate massive amounts of data which can scale up to millions of spectra. The increases in these data rates creates scaling problems for existing standard software designed for much smaller datasets. With the advent of machines such as Thermo Orbitrap Fusion which combine Tribrid architecture, multiple fragmentation techniques, parallelization of MS acquisition, ultrafast resolution and scan rates will ensure a deluge of MS data and will be useful in proteomics, metabolomics, glycomics, lipidomics and similar applications. The raw mass spectrometry data is a combination of mass-to-charge (m/z) ratio and intensity of the peaks and is much more complex than a typical next generation sequencing (NGS) data set. Therefore, data generation is just the first step for useful analysis as phosphopeptide filtering [7], false positive rate estimation [8], quantification of proteins from large datasets [1], and phosphorylation site assignments [9] are some of the essential post-processing steps required.

The usual computational route taken is to search the raw spectra against a protein database. The algorithms used for searching are generally brute force (Sequest, Inspect etc) methods that try to match the spectra to a theoretical spectra in a given database and deduce the peptide sequence. Although these algorithms are useful for interpretation of simple spectra, the search & match routine becomes computationally intractable for complex peptides (e.g. compounded spectra, multiple post-translational modification (PTM's) etc.). Since these algorithms are brute force methods they are not computationally efficient to analyze thousands of spectra in a reasonable time. One solution to efficiently deal with this massive amount of data is to cluster the spectra and combine the clusters to formulate consensus spectra that can be used for further processing.

Clustering of large-scale data is efficient because the peptides usually get selected multiple times in a typical MS-MS run, making a significant portion of the spectra redundant. Searching the same spectra repeatedly wastes both time and computational resources. The problem is even more pronounced when data from multiple runs are merged. The redundancy can reach up to 50% for large data sets [10, 11, 12]. Clustering of spectra from complex biological samples can also increase the sensitivity and confidence in peptide matches [13]. The increase in identifications can be attributed to the fact that clustering allows low S/N spectra to be grouped with high-quality spectra, which in a non-clustered data set would be eliminated from identification. Clustering of MS/MS data is also known to reduce the number of false positive identifications associated with low-quality spectra [10].

By joining together the low and high quality spectra, the probability of making an erroneous identification is greatly reduced as compared to searching the low quality spectra separately. Clustering has its advantages and CAMS algorithm has been shown to be useful in grouping spectra with high accuracy for moderately large data sets. However, enormous reduction in processing time is possible by exploiting multicore architectures to our advantage.

1.1 Contributions

This paper is concerned with effective parallelization of the graph-theoretic framework for clustering of mass spectrometry spectra using multicore architectures. Parallelization of numerous applications using multicore architectures have been successfully pursued in the past [14, 15, 16, 17, 18, 19]. To this end, we introduce an algorithm that exploits both thread-level and instruction-level parallelism on multicore architectures. The design is exible enough to port the algorithm on many-core and extreme multithreaded architectures with the changes in few pragmas and registers. The contributions of the paper can be summarized as follows:

1. A effective algorithm for clustering of mass spectrometry data is devised. The strategy is based on speculation and selective choosing of the nodes that are used to establish clusters. This allows the algorithm to compute the clusters without doing an all-to-all comparison and sets the stage for parallelization strategy of the algorithm.
2. A parallelization strategy, called P-CAMS, suitable for multicore architectures using novel F-set metric and graph theoretic framework is introduced. The strategy first randomizes the order of the spectra and then issues each of the spectra to a distinct thread. The threads works on finding the selected spectra a grouping bin that will allow similar spectra to be grouped together. Finding a bin for the spectra requires the thread to compare the F-sets between two given spectra which is a compute intensive task. We show how instruction level parallelism can be exploited *within* each thread to perform comparisons.
3. The performance of the parallel strategy is studied on a Intel processor based on Nehalem micro-architecture. The performance study is conducted using carefully chosen spectral data, with variable parameters to mimic real world datasets. First the performance is illustrated for thread-level parallelism. Thereafter it is shown that a instruction level parallelism can be exploited within threads and is a powerful combination that can speedup the algorithm by many folds. The results show that the proposed algorithm is highly scalable with increasing size of the datasets and significant time can be saved using our proposed techniques.
4. A quality assessment of the proposed algorithms is done and compared with the sequential version of the algorithms. The data sets used for quality assessment are real data sets generated specifically to assess the quality of the clustering results given by the algorithm.

The rest of the paper is organized as follows: In section 2, we brievely discuss the Nehalem micro-architecture background of the machine used and some benchmarking results. Section 3 gives a very brief background information required for this paper. Section 4 will start with

a brief introduction of the problem and the clustering algorithm introduced in [20]. In section 5 we will describe the design of the algorithm which includes an intelligent matrix completion strategy, OPENMP and SSE(SIMD) implementation details and, a load-balancing strategy. Section 6 discusses the experimental results which include quality assessment using experimental data sets and the scalability of the implemented algorithm. We end in section 7 with discussion and conclusions.

2 The Intel Nehalem

The Intel system we considered is a quadcore Intel(R) Xeon E5620 processor running at 2.40GHz, which is built on Intel Micro-architecture Nehalem. Each core has super-scalar, out-of-order, speculative execution pipelines and the maximum memory bandwidth of the system is 25.6 GB/s. Each core offer multiple functional units which offer high instruction level parallelism and support 2-way simultaneous multi-threading (SMT). An interesting feature of the chip is direct support for integer and floating point SIMD instructions by the hardware at the rate of 4 double or 8 single precision floating point instructions per clock cycle. Each core consists of large number of functional units (FU) and micro-operations can be processed in parallel if there are no dependencies on the results of each other. A parallel strategy that could leverage the usage of hardware based SIMD instructions in an efficient way would be able to exploit massive parallelism. The memory system per core consists of 32KB Instruction cache (Li) and 32KB data cache (Ld). In SMT mode, the cache are shared by hardware threads and the access latency to access data already residing in L1 cache is 4 clocks. Further there is 256KB of L2 cache with access latency of 10 clock cycles. L3 cache is of 12MB shared by eight cores and the access latency is around 35–40 clock cycles. Note that a data dependent element that has to be imported from another core has a higher communication cost since it has to be done through L3 cache. Also note that a data dependent elements that has to be imported from another core in another processor is much higher than either of the previous two costs. Therefore, spatial and temporal locality of the data will have significant impact on the performance of the algorithm.

The Intel system that we considered is a compute server consisting of two quadcore Intel Xeon(R)E5620 processors each running at 2.40GHz. The high level block diagram is shown in Fig. 1. SMT in Nehalem architecture allows more than one hardware thread to execute simultaneously within each core and share the resources; making 8 physical cores with a maximum of 2 threads/core capable of executing 16 threads simultaneously. The system has 60GB of RAM. In order to benchmark the compute system we ran Intel Optimized LINPACK benchmark data. The parameters were set as: number of tests=15, problem size= 1000 to 45000, Leading dimension of array 1000 to 45000, and the number of trials and data alignment values of 4 (maximum). Using the above parameters the system was able to perform at peak performance of 33.9 GFLOPS. The time it took to run the maximum problem size was 1258 seconds. We also ran the STREAM benchmark to assess the memory performance of our system. The results were as follows: Copy rate=3419 MB/s, Scale Rate=3305 MB/sec, Add Rate = 7532 MB/sec, and Triad = 4976.6 MB/sec.

3 Background Information

Mass spectrometry data is inherently stochastic in its nature and requires complex algorithms to process the data into useful information for biologists to use. The usual computational route taken to process the data includes searching the raw data against a known proteome database using search algorithms (e.g. Sequest, Inspect etc.) and assignment of spectra to each peptide. These search engines perform a brute force search to match peptides to spectra which makes these algorithm time and space inefficient. Since there is much redundancy in the data, the search algorithms keep on searching multiple instances of the *same* spectra wasting precious resources. Accurate clustering of mass spectrometry data eliminates the computational inefficiency of the search tools while allows scientists to get more out of the data. Multiple clustering algorithms have been proposed in the past which include, Tabb et. al [21], MS2Grouper algorithm [22], Beer et. al. developed the Pep-Miner algorithm [10], Ramakrishnan et. al. [23], Dutta et. al. [24] and Frank et. al. [12]. More recently a clustering algorithm using graph-theoretic concepts was introduced by Saeed et. al. [20]. The first parallel algorithm for clustering of mass spectrometry data was introduced by Saeed. et. al [25] which exploited thread-level parallelism to speedup clustering using multicore architectures.

In this paper we improve our parallelization strategy to further speedup the clustering process. In order to accomplish this, a hybrid approach is designed to take advantage of streaming SIMD instructions for each of the thread that executes in parallel on independent cores.

4 Problem Statement

Clustering of mass spectrometry data allows one to group spectra that belong to the same peptide i.e. each spectrum would map to the same peptide if it were searched separately using a search algorithm such as Sequest or Mascot. Clustering occurs before any spectral searching i.e. the peptides that correspond to the spectra are not known when clustering is accomplished. The objective of the clustering is formally defined below. The reader is referred to [20] for detailed discussions:

Definition 1 *Let there be N number of spectra $S = \{s_1, s_2, \dots, s_N\}$. Now let the proteins be represented as string over the amino acid alphabet and peptide is a fragment obtained from the protein. Let the total number of distinct peptides under consideration be represented as $P = \{p_1, p_2, \dots, p_N\}$. Now let the peptide corresponding to a spectra s_q represented by p_q where $1 \leq q \leq N$.*

Definition 2 *A distance function $\delta(p_r, p_t)$ where $p_r \in P, p_t \in P$ is defined as the levenstein distance of the peptides corresponding to the spectra s_r and s_t . Now let the number of clusters be k and represented as $K = \{k_1, k_2, \dots, k_k\}$ such that set S is divided into k subsets. Then, the spectra s_r and s_t where $s_r, s_t \in S$ should belong to the same cluster k_i where $k_i \in K$, if and only if, $\delta(p_r, p_t) = 0$ where $p_r, p_t \in P$. Some error tolerance in δ distance can be introduced for miscleaved peptides.*

4.1 Overview of the serial clustering algorithm

We have introduced a novel metric, called F-set, that gives accurate clustering results as compared to single peak comparison between spectra. The basic idea for using the F-set metric is to use groups of consecutive spectral peaks instead of using single peaks for comparing spectra. Using consecutive group of peaks instead of single peaks eliminates the possibility of matching random peaks due to noise and leads to more accurate comparison metric for the clustering algorithm. The formal Definition for the F-set metric as stated in [20] is as follows:

Definition 3 As before, let the spectral data set be represented as $S = \{s_1, s_2, \dots, s_N\}$. Each spectra has two attributes i.e. m/z and the intensity of the peak. Let there be a fragmentation spectrum $s_j = (m_1, i_1), (m_2, i_2), \dots, (m_Q, i_Q)$ that is extracted from the mass spectrometry data where m_d represents the m/z ratio and i_d represents the intensity of the peptide at position d and $1 \leq j \leq N$.

Now making sets of peak's at m/z positions of size f . Then creating sets out of the spectra can be presented as a vector $F(s_i) = \{(m_1 m_2 \dots m_f), (m_2 m_3 \dots m_{f+1}), \dots, (m_{Q-f+1}, \dots, m_{Q-1}, m_Q)\}$. Then the F-set metric calculated for spectra s_x and s_y can be formulated as

$$W(s_x, s_y) = \sum_{i=1}^{|F(s_x)|} \sum_{j=1}^{|F(s_y)|} \phi(F(s_x)[i], F(s_y)[i]) \quad (1)$$

$$\phi(a, b) = \begin{cases} 1 & \text{if } a=b \\ 0 & \text{o.w.} \end{cases} \quad (2)$$

The F-set, denoted by $W(s_x, s_y)$, can be used as a similarity metric for spectra. F-set metric makes sets of peaks of size f and then compares them to other set of peaks from other spectra. A scoring scheme can be formulated according to the spectra being considered. As a proof of concept we have devised a very simple scoring mechanism. If there is a match of a F-set in the other spectra, a score of 1 is added and there is zero penalty for non-matching F-sets. The final score W represents the number of F-sets that are common between two given spectra when using the scoring scheme illustrated above. Other scoring mechanisms may have different meaning according to the spectra being considered.

In order to efficiently process the clustering strategy we defined a graph-theoretic framework that allows CAMS algorithm to take advantage of the similarity metric. A weighted undirected graph is created where the vertices represent each of the spectra and the weighted edges represent the similarity of the two connected vertices (spectra). With our scoring scheme, the higher the weight on the edge, the more similar the connected spectra. The stochastic nature of the mass spectrometry data dictates that even though we use sets of peaks to make our comparisons, there will be some sets that will be common between two unrelated spectra. These common sets can arise from compounded spectra, from spectra that may have few regions common between two given peptide or just because of noise. In our experiments we observed that for spectra that were not related will accumulate a small score.

Although, the weight for non-related spectra was much smaller than for the spectra that were related, the algorithm needed a way to distinguish between the two. Therefore we define a threshold called ζ to remove the edges that have a weight less than the specified threshold. The vertices that are still connected via an edge are considered to be clustered together. Our controlled experiments have suggested that the weight of more than 30 where F-set size is 7 or more is sufficient to accurately group related spectra. The formal Definitions are given below:

Definition 4 A weighted undirected graph $G = (V, E)$ is a graph where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. Now let a weight $w_e = (v_i, v_j) \geq 0$ associated with edge $e = (v_i, v_j)$ where $e \in E$ and $v_i, v_j \in V$. Now given a graph $G = (V, E)$ such that the number of vertices in the graph are equal to the number of spectra being considered i.e. $|V| = |S| = N$ and an edge connecting each vertex. Now vertices can be represented by $V = v_1, v_2, \dots, v_N$. Then, the nodes can be labeled using the following mapping function $\forall v_i \rightarrow s_i$ where $v_i \in V$, $s_i \in S$, $1 \leq i \leq N$. The weight on each edge is the F-set metric that is calculated for the spectra i.e. $w_e = W(s_i, s_j)$ where $e = (v_i, v_j)$; $s_i, s_j \in S$; $e \in E$, $v_i, v_j \in V$.

The reader is referred to [20] for details about the CAMS algorithm and is not restated here in the interest of brevity.

5 Multithreaded approach to CAMS algorithm

We have made few fundamental changes in the design of the CAMS algorithm in order to make it more efficient and have structured the changes in a way that helps in efficient parallelization on multicore architectures.

5.1 Intelligent Matrix filling

The crux of the algorithm is the F-set metric that has to be calculated between two given spectra. This weight is then used to form weighted edges for the graph. In order to build the graph, an adjacency matrix has to be filled using the F-set metric Definition discussed in the paper. However, calculating this score is computationally expensive and an all-to-all comparison makes the algorithm intractable with increasing number of spectra. We wanted to investigate if we could do the same without computing all the combinations of F-set metrics for each of the spectra under consideration i.e. if we could fill up the adjacency matrix more intelligently so that all the possible combinations are not computed but *same* clustering results are achieved.

The basic idea for filling up the matrix in an intelligent way is the observation that we do not need to do an all-to-all comparison to fill up the matrix i.e. if a vertex (which corresponds to a single spectra) is already a part of a cluster group then there is no need to further compute the edges related to this vertex. In other words, if a vertex has been added to any of the cluster trees, then we do not need to compute any more F-set metric related to that vertex. This restriction allow us to get graph structures that are tree-like instead of graph structures with closed loops, or other complex patterns. Looking at figure 2 it can be seen that when traversing the first row of the matrix, vertices B,C,D are added to the cluster tree (vertex A). This is indicated by red numbers in the matrix and red edges in the corresponding graph that

have been connected due to the weight of these edges greater than the threshold $\zeta = 30$ (for this example). Now the other comparisons such as between (B,C), (B,D) and (C,D) are redundant because they do not add any information to our existing knowledge about the spectra i.e. (B,C,D) should be added to the graph that has vertex A. Using this intelligent matrix filling technique, the code can avoid expensive spectral comparisons that are redundant. Note that for the example presented in the figure, the computational saving using intelligent matrix filling is equal to 20% and the savings increase rapidly with increasing number of spectra.

5.2 P-CAMS: Parallel CAMS algorithm

The intelligent matrix filling technique allows us to eliminate comparison without explicitly comparing the spectra. The scheme therefore allows enormous advantage in saving precious CPU cycles. The all-to-all comparisons are reduced by a factor of percentage of cluster that is present in the data. The strategy also allows effective parallelization because vertices can be randomly selected and compared against the vertices that are available in the pool in parallel. The modified CAMS algorithm is stated in Algorithm 1 and 2.

Only the part of the algorithm that is relevant to the parallelization is presented in the paper. Algorithm1 shows how we generate the F-set as defined in the paper. The algorithm shows that the m/z of the spectra is extracted from the .dta files from raw data and made into a vector. This vector V is read and F-sets of specified size are generated. A sentinel key is also defined to divide the sets that are defined. Algorithm 2 is the most compute intensive method in the CAMS algorithm. It shows that two vectors are extracted and each element of the vector is compared with every other element. The outer loop in the algorithm 2 can be executed in parallel since each element from vector1 can be compared to all elements from vector2 in parallel. Since excessive parallelization can negatively effect the performance, only the outer *for* loop is done in parallel so that cache contention does not take place. Algorithm2 takes in two vectors that contain F-sets. These vectors are used to compare the F-sets and return the number of sets that are common between the two.

The Algorithm 3 illustrates the intelligent matrix completion strategy and requires some explanation. The algorithm starts with an array names *Fsetarray* and is filled using the *generateFset* method illustrated in Algorithm1. The *Fsetarray* is filled up with *Fsets* for each of the spectra. After the *Fsets* have been generated and filled up the *Fsetarray*, the intelligent matrix filling and connecting of the edges is initiated. The ordering of the spectra can adversely effect the running times of the algorithm. We randomize the ordering of the spectra to eliminate any effects the ordering may have on the comparison or the running times of the algorithm. We will also discuss this in some detail in the load-balancing part

Algorithm 1

GenerateFsets (Vector V , int size)

Require: Vector of spectra containing m/z (vector V , int size)

Ensure: Generates Fsets from a vector V of m/z of stated size

$Fset \leftarrow \emptyset$


```

for ( $i = 0; i < V.size(); i++$ ) in parallel do
  if ( $i + size \leq V.size()$ ) then
    for ( $j = 0; j < size; j++$ ) do
       $temp \leftarrow V[i]$ 
       $Fset = Fset + temp$ 
       $i++$ 
    end for
     $i = i - size$ 
     $Fset = Fset+;$  // ; is used as a sentinel character to distinguish between sets of Fsets
  end if
end for
return  $Fset$ 

```

Algorithm 2

comparekmeriset (string first-string, string second-string)

Require: Pair of F-set strings generated from m/z of spectra using GenerateFsets function

Ensure: Returns the number of F-set that are common between the two spectra

/* splitfunction(string X) takes a string X and changes it into vector where each entry in vector corresponds to a single F-set */

vector firstv=splitfunction(first-string)

vector secondv=splitfunction(second-string)

int count=0

for ($i = 0; i < firstv.size(); i++$) **in parallel do**

for ($j = 0; j < secondv.size(); j++$) **do**

if ($firstv[i].compare(secondv[j]) == 0$) **then**

 count++;

end if

end for

end for

return count;

Algorithm 3

Cluster Spectra (N spectra, Fsetsize)

Require: N Spectra, Fsetsize

Ensure: Cluster the spectra and output the connected components

$Fsetarray[] \leftarrow \emptyset$

$filearray[]$ //Each entry in filearray contains m/z of a single spectra

for ($k = 0; k < N; k++$) **in parallel do**

$Fsetarray[k] = GenerateFsets(filearray[k], Fsetsize);$

end for

// randomize the ordering of N

```

for ( $i = 0; i < N; ++i$ ) in parallel do
  if ( $Fsetarray[i] \neq \emptyset$ ) then
    for ( $j = i + 1; j < N; ++j$ ) do
      if ( $Fsetarray[i] \neq \emptyset \ \&\& \ Fsetarray[j] \neq \emptyset$ ) then
        if ( $comparekmersset(Fsetarray[i]; Fsetarray[j]) \geq threshold$ ) then
          //draw an edge between spectra named filearray[i] and filearray[j] in Graph G
           $e(filearray[i], filearray[j])$ 
           $Fsetarray[j] \leftarrow \emptyset$ ;
        end if
      end if
    end for
  end if
   $Fsetarray[i] \leftarrow \emptyset$ 
end for
return connected components of Graph G

```

of the paper. Note that the second *for* loop is only initiated if the $Fsetarray[i]$ has not be invalidated, denoted by $\emptyset$. The invalidation of the entry occurs when the said node has been added to any of the clusters. Likewise the comparisons only takes place if both of the entries in the $Fsetarray$ i and j are valid. If both the $Fsetarray$ entries are valid then the most expensive *comparekmersset* method (algorithm 2) is executed to perform the comparisons. Algorithm 2 returns the number of $Fsets$ that are common between the two given spectra. The returned value is then used as a weight on the edge and as a measure of similarity i.e. greater the weight the more similar the two given spectra. One of the vertex (the inner *for* loop object) is then added to the bucket and eliminated so that no more comparison takes places for this particular vertex. At the end of the algorithm the connected components from graph G are reported, where G is a forest. Each tree formed in graph G is considered a cluster of spectra. Note that the components that are connected would have greater probability that they are related to each other (i.e. belonging to the same peptide) due to large number of F-sets that are common between them.

5.3 OPENMP Implementation

Now we discuss the OPENMP implementation of the improved P-CAMS algorithm. The implementation of the algorithm was done using gcc version 4.4.6 20110731 (Red Hat 4.4.6-3) and Linux Scientific version 2.6.32 – 220.13.1.el6.x8664 as the operating system. Please note that only main loops and computationally intensive parts of the code will be illustrated. OPENMP implementation will allow us to exploit thread-level parallelism for our algorithm.

5.3.1 Counting common F-sets between spectra—This section of the code is about counting the number of F-sets that are common between the two given spectra. This translates into comparing two vectors of string and counting the number of substrings that are common between the two. This part is also most compute intensive part of the algorithm

since this has to be done for all spectra and each spectra F-sets have to be compared with that of every other spectra.

The design of the algorithm 2 suggests that we pick an element from the first vector and compare it with all the elements in the second vector and this can be done in parallel i.e an element from the first vector can be compared with all the elements in the second vector simultaneously with the first iteration. This should make it a highly parallelized version of the clustering algorithm. The OPENMP code is implemented in this way that each thread can pick up an element from first vector and compare it all the elements in the second vector and each thread will be able to execute simultaneously in parallel. For the code shown in Fig. 3 it is assumed that the vectors that are taken as input have the *Fset* in place.

In order to facilitate the counting procedure from two vectors a `#pragma omp parallel for with the reduction(+:count)` is used. The `reduction(+:count)` clause performs a reduction on the variables that appear on the list, which in our case is the integer count. Using this clause a private copy of count is created for each thread and at the end of the reduction, count variable is applied to all the private copies of the shared variable and the final results are written to the globally shared variable. In this way the counting process becomes highly parallelized.

5.3.2 Parallelizing the Main Loop in Algorithm 3—Parallelization of main loop for algorithm3 using OPENMP pragmas is shown in Fig. 4. The total number of spectra that have to be clustered is represented by `TOTAL_ITEMS`. The `kmer_set` is an array which contains the *Fsets* from the spectra created from the earlier step and the contents of the array is then used to compare the *Fsets* between given spectra. If the spectrum becomes a dont care condition according to our intelligent matrix filling, we denote it with an `xx`. There are two methods used in the code shown. `mystring()` return a string `xx` and the method `comparekmeriset(string x, string y)` returns the number of F-sets that are common between two given strings (Algorithm 2).

The code shows that the outer loop is parallelized using `#pragma omp parallel` where the array `kmer_set` is the global array that is shared among all threads. The variables `i` and `j` are kept private and each thread get its own copy. The global array `kmer_set` can be updated at the outer `for` loop that happens each time an iteration of the inner loop is completed and the other update is in the inner `for` loop that updates the array whenever a comparison from the method `comparekmeriset` returns a weight greater than the required threshold. Since, the array `kmer_set` is global in its nature, it is necessary to make `#pragma omp flush` to give the other threads a coherent view of the global array. However, the `ush` update is an expensive process. Therefore, we opted for a row-wise update of the global array i.e. the update for the global array is done once all the iterations in the inner loop are completed. Therefore, other threads will see the dont care condition by the inner loop once iteration has completed. However, explicit `ush` is not necessary in our case since an `ush` is implied when exiting the `for` loop.

Note that we implemented the algorithm using arrays instead of stacks and lists. The reason is increased running time for the later two which can be attributed to time consumed in

deleting the linked list nodes and reinserting stack values. It is also more convenient to deal with arrays in OPENMP and Cray XMT implementation instead of linked lists or stacks which require an explicit iterator to go through their structures.

5.4 Exploiting Instruction-level parallelism within each Thread

Profiling of our OPENMP program made it clear that comparing of F-sets between spectra, implemented using `comparekmerset` method, was the most compute intensive part of the algorithm. Intel Simultaneous Multi-Threading technology (SMT) allows more than one hardware thread that can be executed on each of the core. For 8 physical core system with 2 threads per core the machine should be able to execute 16 threads in parallel. With OPENMP implementation we observed decrease in running times with increasing number of threads but the parallelism was not enough i.e. with increasing threads on a dedicated machine we were not able to see results as expected. Further it was clear that more than 4 threads was detrimental to the performance of the algorithm (see results section).

This parallel slowdown suggested that the processing is not taking place as fast as it should and cache contention is taking place making the running time slower with increasing threads after a certain point. Although, this is true for any parallel program, the machine and the algorithmic analysis suggested that we should expect speedups by a factor of 16 for the hardware being considered. Also, in ideal conditions the running time of the parallel algorithm must not be effected by an arbitrary increase in the number of threads. It was clear that with increasing threads the cores were not able to complete the comparison tasks fast enough for the remainder of the threads to process in parallel. Increasing threads leads to slower run-times due to the increases in context switch times, cache congestion/contention and variety of bandwidth issues.

One solution to this problem was to make the threads perform the required task in shorter time which will increase the system bandwidth. To this end, we take advantage of SIMD instruction set in Intel Neehlam architecture. SIMD processing can be performed using Intel SSE4.2 instruction set that also allows string comparisons and has direct support for integer and oating point instructions by the hardware. The SSE instruction allows us to perform four comparison in parallel instead of single oating point comparison. This allows us to exploit instruction level parallelism *within* each thread. The high-level diagram illustrating the parallelism is shown in Fig. 5. Recall that the original intention of the method was to compare two vectors which contained F-sets. For the sake of simplicity we assume that the size of F-set is 1 which makes vectors that need to be compared have single numbers. In order to exploit data parallelism in this we divide (logical) the array B into sets of 4 since each threads will be able to do 4 comparison in parallel using SSE hardware instructions. Now four elements from array A are compared with 4 elements in array B in parallel. In order to make the comparisons complete, the 4 elements in array B are shuffled and the comparison is performed again with the original 4 elements of array A. This is done 4 times for each four-element subset of array B and is done for all subsets. The procedure is illustrated in Fig. 5 and the code is shown in Fig. 6. Note that there is no OPENMP instruction in the code. This is because the `comparekmerset` is called by threads in each loop iteration shown in Fig. 4 and hence allows us to realize tremendous parallelism using

multicore architecture. Also note that all of the shuffling comparison can be performed in parallel.

5.5 load balancing

Load balancing is one of the most important attributes necessary for performance of a parallel algorithm [26]. Load-balancing ensures that the processors/cores are busy most of the time simultaneously. Without a balanced load, some of the threads may finish much later than others, making the running time of the program bounded by the last thread that completes. Computational load of a single thread/core cannot be predicted before the actual execution. However, the specific nature of the mass spectrometry data under consideration will let us craft a load-balancing scheme that allows near equal loads on each core.

We use specific characteristics of mass spectrometry data to formulate our load-balancing scheme. The data from the mass spectrometer is ejected so that peptides that have similar m/z ratio would come out in groups. Our intelligent matrix scheme is based on the observation that the vertices that are clustered do not need to be compared thereafter. This means that if more similar elements are present on the same core, then that core will be done with comparisons much more quickly than others; leading to un-balanced load on the cores. Fig. 7 shows the scenario for an unbalanced load (upper array). As can be seen that if the data is distributed in a trivial way then the number of don't-care-conditions will increase more rapidly for cores that have more similar spectra than the cores that have fewer number of spectra that originate from the same peptide.

We will show that a counter-intuitive approach which will distribute similar spectra on distinct cores will make the computations more load-balanced. We redistribute the spectra on different cores which is shown at lower end of the array in the Fig. 7. Now when comparisons for spectra a are completed, the resulting load-reduction occurs on cores 2 and 3. The load-reduction is uniform which leads to more balanced load-per-core for the algorithm. The load is therefore distributed in a round-robin fashion over the number of cores which gives enough randomization for the data but keeping the number of spectra approximately equal on each of the core. This corresponds to *static* scheduling in OPENMP code. In order to test if the technique works correctly we took 2000 and 4000 spectra and applied various scheduling combinations. The results of applying these scheduling techniques can be seen in Fig. 8. The scheduling techniques applied were 1) dynamic scheduling 2) ordered loops with dynamic scheduling 3) ordered scheduling and 4) static scheduling with different chunk sizes. As is clear from the figures that the first three scheduling techniques have adverse effects on the running times of the algorithm with increasing number of threads. The performance degrades further with increasing data set size as can be seen for the performance of 4000 spectra. The trend of poor performance continues with increasing number of spectra. The performance of static scheduling is very consistent with increasing number of threads as well as with increasing data set size. The chunk size seems to have minimal effect on the static scheduling scheme.

6 Experimental Results

We chose the CID and HCD data sets that we had used in our earlier paper [27] to assess the quality of the clustering algorithm. The experimental protocol used for generating the data is stated here very briefly. A piece of freshly isolated rat liver was minced and sonicated in guanidine-HCl (6M, 3ml). A peptide standard corresponding to the C-terminal sequence of the water channel Aquaporin-2 (AQP2) from rat, (Biotin-LC-CEPDTDWEEREVRRRQS*VELHS*PQSLPRGSKA) phosphorylated at both S256 and S261 were added to 500 µg aliquots of liver sample (prior to trypsinization) with distinct amounts of 0.2 nmol, 20 pmol and 2 pmol and were named AQP2-H-(S256/S261), AQP2-M-(S256/S261), AQP2-L-(S256/S261) respectively. The same procedure as above was repeated for another AQP2 peptide standard (Biotin-LC-CEPDTDWEEREVRRRQSVLHSPQS*LPRGSKA) phosphorylated at S264, with amounts of 0.2 nmol, 20 pmol and 2 pmol and were named AQP2-H-(S264), AQP2-M-(S264), AQP2-L-(S264) respectively. Peptide samples were desalted, and then were resuspended in 0.1% formic acid prior to analysis by mass spectrometry using HCD as well as CID fragmentation. The mass spectrometry raw data was then processed by the Sequest search engine to identify the peptides and used as a ground-truth data set for the quality assessment of the proposed algorithm. Please note that although we have used synthetic peptides, the data also contains spectra corresponding to other peptides in the rat liver.

The scalability of the algorithm has to be tested by using large data sets. In order to be comprehensive and to make sure that the data used for scalability assessment is close to real-world data sets we implemented a spectra generator. The basic idea of the generator is to specify percentage of the spectra that are to be clustered, and generating the rest of the spectra randomly. This way we would get spectra that are close to the real-world data sets. We generated 30% of the data that comprised of similar spectra (which would potentially be clustered together) and 70% of the data is randomly generated for a given number of spectra. This ensures that at least 30% of the spectra will need to be clustered by the algorithm with increasing data set size. The percentage of data set that will be clustered in real-world data will be much less than 30%.

6.1 Scalability results with OPENMP Implementation

In order to assess the scalability of the proposed algorithm we tested the OPENMP implementation. Note that for this section SIMD instructions are not used in the code. The CPU times and the associated speedups can be seen in Fig. 9 where CPU times drop sharply with increasing number of threads. The times decrease sharply and the lowest times are observed for 4 threads which is equivalent to the number of physical cores in the system. For up to 4 threads the time decreases sharply and even super-linear speedups are observed for most of the data sets. However, utilizing more than 4 threads *increases* the execution time and adversely affects the scalability of the algorithm. The reason for this slowdown of the program can be attributed to cache congestion since more number of threads are trying to access the data simultaneously resulting in loss of locality of the data for the immediate cache. Other reasons are increase in the context switch times and bandwidth issues. Increasing number of threads do not seem to speedup the process beyond a certain point

because the threads are not done performing the comparisons before other threads have reached the comparison stage and compete with each other for core resources such as fast-cache access. We will show how the slowdown of the parallel algorithm can be mitigated by speeding up the comparison process using SIMD instructions. Although, the results from this parallelization efforts were not as expected, the experiments did serve us to show that speedups are observed due to the introduction of intelligent matrix and parallelization effort and the behaviour is consistent with increasing number of data sets.

6.2 Scalability results with OPENMP and SIMD Implementation

Although substantial reduction in timing were observed with OPENMP implementation it is clear that the speedups were limited by the number of threads that can be utilized. The bottleneck in the performance of the algorithm were two factors 1) the threads were not able to complete tasks rapidly enough and increasing number of threads compete with each other for core and system resources 2) the reason that threads were not able to process the comparisons fast enough, as dictated by the architecture being used (i.e. 16 simultaneous threads), is that OPENMP does not take advantage of the cores at the instruction level. So OPENMP allows one to issue large number of threads and large number of threads would clog the L3 cache. It was also clear that spatial and temporal locality of the data could not be improved using OPENMP alone. If a way could be devised that will allow more rapid comparison, the incoming threads will have more hardware at their disposal and L3 cache will not be clogged.

Profiling of our program suggested that `comaprekmerset` method was the most compute intensive method where algorithm spent most of its time. To perform these comparison in parallel on each available core we devised the SIMD formulation of the algorithm that could take advantage of the instruction set hardware as described in the section 5.4. The results with increasing data set size and increasing number of threads is shown in Fig. 10. As can be seen that with increasing number of threads the execution time keeps on decreasing and super-linear speedups are observed. This is because each of the thread is able to take advantage of data parallelism offered by SIMD instructions which allow 4 consecutive comparisons on *each* core. For a 16 core system, the number of comparison that can be performed in parallel would be $16 \times 4 = 64$ *simultaneous* comparisons. This shows that instruction-level parallelism can be successfully exploited *within* each thread for comparisons on multicore architectures. The scalability of the algorithm is unaffected with increasing data set size.

6.2.1 Scalability with increasing number of F-set size and number of threads—

The complexity analysis of the algorithm suggest that the running times depends on both the size of the F-set and the number of spectra being clustered. We wanted to investigate how scalable is the parallel algorithm with increasing size of F-sets. The sizes of F-sets used are 2,4,8,10,12 and the number of spectra was increased up to 32000. The results of the program with increasing number of threads can be seen in Fig. 11. As can be seen that with increasing number of spectra, the run times decrease very sharply with increasing number of spectra as well as with increasing F-set size. Note that we have previously shown that F-set

size of around 7 is more than sufficient for accurate clustering of high-resolution mass spectrometry data [20].

6.3 Quality Assessment

The quality of the clusters were assessed using the data sets just described and the parameters that were defined for assessing the quality in an accurate manner. The average weighted accuracy (AWA) allows us to assess the intra- as well as inter- accuracy between clusters [20]. The metrics are defined here again for completeness. Assume there are k clusters. Now let the accuracy of a single cluster i be denoted by a_i and the total number of spectra in the cluster be defined as n_i where $1 \leq i \leq k$. Now assume that the number of spectra in a cluster that belong to the same peptide be denoted by x_i . Then, the accuracy of a *single* cluster can be defined as

$$a_i = \frac{x_i}{n_i} \quad (3)$$

and the average weighted accuracy (AWA) of the whole dataset under consideration is defined as:

$$AWA = \frac{\sum_{i=1}^k a_i n_i}{\sum_{i=1}^k n_i} \quad (4)$$

AWA takes into account the accuracy of each cluster and gives a global view of the accuracy for a given dataset. The spectra to peptide matching has been done using Sequest for the data sets in our experiments. This apriori information allows us to do a accuracy assessment of the clustering algorithm. The advantages of using F-set for clustering is shown to improve the AWA with increasing size of the F-set [20].

The accuracy for the datasets is reported in table 1 using P-CAMS. As is reported in the table, P-CAMS gives comparable quality clusters for both CID and HCD data sets and the accuracy of the clusters are very close to 100%. The clustering is performed using P-CAMS and the clustered spectra are then compared with the results from Sequest i.e. the spectra that are clustered must have the same peptide when Sequest is used to search the spectra. We also compared the results from P-CAMS with other algorithms such as MS-Cluster and SPECLUST. Our experiments suggest that the clusters obtained using P-CAMS are more accurate and precise than either of these tools (results not shown).

In order to make the quality assessment more comprehensive we also used Proteomics Dynamic Range Standard Set (UPS2) from Sigma-Aldrich to test our strategy. UPS2 can be used to standardize and evaluate LC-MS/MS and electrophoretic analysis conditions. The UPS2 contains a mixture of 48 individual human sequence recombinant proteins, each of which has been selected to limit heterogenous post-translational modifications. The details experimental protocol can be seen in [13]. The increase in the confidence of the peptide match is shown in table 2. As can be seen in the table that increasingly stringent criteria of clustering allows more confident matches. Hence clustering can be used for compression of large data sets as well for increasing the confidence of spectra-to-peptide matches.

7 Discussion and Conclusion

Clustering of mass spectrometry data has advantages in post-processing procedures in terms of compression of large data sets as well as increasing confidence of the peptide identifications. Clustering also helps to identify peptides and proteins that are low in abundance and low quality (low S/N ratio) spectra. The next generation of mass spectrometry instruments are capable of producing highly accurate spectra in large number in short amounts of time. High performance techniques are essential to analyze these large data sets.

The objective of this work was to provide a tool to efficiently handle and interpret large scale mass spectrometry data sets. In this paper we present a technique to cluster large-scale mass spectrometry data by exploiting instruction-level parallelism *within* each thread on multicore architectures. The clustering algorithm, called P-CAMS, is based on a novel similarity metric we call F-set. The similarity metric is based on comparison of groups of peaks instead of single peaks and is more accurate than single peak comparison. A graph-theoretic framework is utilized to take advantage of F-set metric for clustering. The parallelization of the algorithm is accomplished by using an intelligent graph-theoretic scheme that allows tremendous reduction in the possible comparisons which can be performed in parallel using multiple threads. Further parallelization is achieved by using SIMD instructions within each thread that allows us to exploit massive parallelism using a moderate multicore machine. This instruction level parallelism exploited within each thread allows decrease in running times with increasing number of threads far beyond the number of physical cores. A load-balanced scheme is introduced that allows efficient comparisons between spectra on multiple cores. The technique allows enormous advantages in time reduction and super-linear speedups are observed with increasing number of threads. Experimental data sets using real mass spectrometry techniques were generated for assessing the quality of the clustering algorithm. We report very high quality clusters using the algorithm.

There is much talk about multicore architectures and their usefulness for big data analysis. However techniques that can efficiently utilize these manycore architectures to their fullest and formulation of highly scalable algorithms for these architecture(s) remain a significant challenge. We show that a deep understanding of the high performance architectures is essential for designing and implementing parallel algorithms for big data problems. A good design that can take full advantage of parallel processing capabilities of the architecture is essential for scalable solutions to big data problems.

References

1. Hoffert J, Pisitkun T, Wang G, Shen F, Knepper M. Quantitative phosphoproteomics of vasopressin-sensitive renal cells: regulation of aquaporin-2 phosphorylation at two sites. *Proc. Natl. Acad. Sci. U.S.A.* 2006; 103(18):7159–7164. [PubMed: 16641100]
2. Li X, Gerber SA, Rudner AD, Beausoleil SA, Haas W, Elias JE, Gygi SP. Large-scale phosphorylation analysis of alpha-factor-arrested *saccharomyces cerevisiae*. *J Proteome Res.* 2007; 6(3):1190–1197. [PubMed: 17330950]

3. Gruhler A, Olsen JV, Mohammed S, Mortensen P, Fargeman NJ, Mann M, Jensen ON. Quantitative Phosphoproteomics Applied to the Yeast Pheromone Signaling Pathway. *Molecular & Cellular Proteomics*. 2005 Mar.4:310–327. [PubMed: 15665377]
4. Whitelegge JP. Hplc and mass spectrometry of intrinsic membrane proteins. 2003 Dec.251
5. Cantin T, Venable D, Cociorva D, Yates R. Iii quantitative phosphoproteomic analysis of the tumor necrosis factor pathway. *J. Proteome Res*. 2006; 5:127. [PubMed: 16396503]
6. Beausoleil A, Jedrychowski M, Schwartz D, Elias E, Villen J, Li J, Cohn A, Cantley C, Gygi P. Large-scale characterization of hela cell nuclear phosphoproteins. *Proc. Natl. Acad. Sci. U.S.A.* 2004; 101:12130. [PubMed: 15302935]
7. Jiang X, Ye M, Han G, Dong X, Zou H. Classification filtering strategy to improve the coverage and sensitivity of phosphoproteome analysis. *Analytical Chemistry*. 2010; 82(14):6168–6175. [PubMed: 20568719]
8. Du X, Yang F, Manes NP, Stenoien DL, Monroe ME, Adkins JN, States DJ, Purvine SO, Camp DG II, Smith RD. Linear discriminant analysis-based estimation of the false discovery rate for phosphopeptide identifications. *Journal of Proteome Research*. 2008; 7(6):2195–2203. [PubMed: 18422353]
9. Ruttenberg BE, Pisitkun T, Knepper MA, Hoffert JD. PhosphoScore: an open-source phosphorylation site assignment tool for MSn data. *Journal of proteome research*. 2008 Jul.7:3054–3059. [PubMed: 18543960]
10. Beer I, Barnea E, Ziv T, Admon A. Improving large-scale proteomics by clustering of mass spectrometry data. *PROTEOMICS*. 2004; 4(4):950–960. [PubMed: 15048977]
11. Frank AM, Bandeira N, Shen Z, Tanner S, Briggs SP, Smith RD, Pevzner PA. Clustering Millions of Tandem Mass Spectra. *Journal of Proteome Research*. 2008 Jan.7:113–122. [PubMed: 18067247]
12. Frank AM, Bandeira N, Shen Z, Tanner S, Briggs SP, Smith RD, Pevzner PA. Clustering millions of tandem mass spectra. *Journal of Proteome Research*. 2008; 7(1):113–122. [PubMed: 18067247]
13. Saeed F, Hoffert JD, Knepper M. Cams-rs: Clustering algorithm for large-scale mass spectrometry data using restricted search space and intelligent random sampling. *IEEE Transactions on Computational Biology and Bioinformatics*. 2013
14. Catalyurek UV, Feo J, Gebremedhin AH, Halappanavar M, Pothen A. Graph coloring algorithms for multi-core and massively multithreaded architectures. *Parallel Computing*. 2012; 38(1011): 576–594.
15. Majumder T, Borgens M, Pande P, Kalyanaraman A. On-chip network-enabled multicore platforms targeting maximum likelihood phylogeny reconstruction. *Computer-Aided Design of Integrated Circuits and Systems*, *IEEE Transactions on*. 2012 Jul.31:1061–1073.
16. Liu Y, Schmidt B, Maskell D. Parallelized short read assembly of large genomes using de bruijn graphs. *BMC Bioinformatics*. 2011; 12(1):354. [PubMed: 21867511]
17. Riedy, J.; Meyerhenke, H.; Bader, D.; Ediger, D.; Mattson, T. Analysis of streaming social networks and graphs on multicore architectures. *Acoustics, Speech and Signal Processing (ICASSP, 2012 IEEE International Conference on; IEEE; 2012. p. 5337–5340.*
18. Sarje A, Zola J, Aluru S. Accelerating pairwise computations on cell processors. *Parallel and Distributed Systems*, *IEEE Transactions on*. 2011 Jan.22:69–77.
19. Ozyer T, Alhadj R. Parallel clustering of high dimensional data by integrating multi-objective genetic algorithm with divide and conquer. *Applied Intelligence*. 2009; 31(3):318–331.
20. Saeed, F.; Pisitkun, T.; Knepper, MA.; Hoffert, JD. An efficient algorithm for clustering of large-scale mass spectrometry data. *Bioinformatics and Biomedicine (BIBM, 2012 IEEE International Conference on; IEEE; 2012. p. 1–4.*
21. Tabb DL, MacCoss MJ, Wu CC, Anderson SD, Yates JR. Similarity among tandem mass spectra from proteomic experiments: detection, significance, and utility. *Analytical Chemistry*. 2003; 75(10):2470–2477. PMID: 12918992. [PubMed: 12918992]
22. Tabb DL, Thompson MR, Khalsa-Moyers G, VerBerkmoes NC, McDonald WH. Ms2grouper: Group assessment and synthetic replacement of duplicate proteomic tandem mass spectra. *Journal of the American Society for Mass Spectrometry*. 2005; 16(8):1250–1261. [PubMed: 15979332]

23. Ramakrishnan SR, Mao R, Nakorchevskiy AA, Prince JT, Willard WS, Xu W, Marcotte EM, Miranker DP. A fast coarse filtering method for peptide identification by mass spectrometry. *Bioinformatics*. 2006; 22(12):1524–1531. [PubMed: 16585069]
24. Dutta D, Chen T. Speeding up tandem mass spectrometry database search: metric embeddings and fast near neighbor search. *Bioinformatics*. 2007; 23(5):612–618. [PubMed: 17237061]
25. Saeed F, Hoffert JD, Knepper M. A high performance algorithm for clustering of large-scale protein mass spectrometry data using multi-core architectures. *International Symposium on Network Enabled Health Informatics, Biomedicine and Bioinformatics (HI-BI-BI)*. 2013
26. Saeed F, Khokhar A. A domain decomposition strategy for alignment of multiple biological sequences on multiprocessor platforms. *Journal of Parallel and Distributed Computing*. 2009; 69(7):666–677.
27. Saeed, F.; Pisitkun, T.; Hoffert, JD.; Wang, G.; Gucek, M.; Knepper, MA. An efficient dynamic programming algorithm for phosphorylation site assignment of large-scale mass spectrometry data. *Bioinformatics and Biomedicine Workshops (BIBMW)*, 2012 IEEE International Conference on; IEEE; 2012. p. 618-625.

Figure 1.

The architecture of a multicore server that we used for parallelization of CAMS clustering algorithm is shown. The machine used has 8 cores and each core can run a maximum of 2 threads. Each of the cores has a L1 and L2 cache and a shared L3 cache.

	A	B	C	D	E	F
A	X	40	51	32	4	16
B	X	X	72	21	17	7
C	X	X	X	43	1	21
D	X	X	X	X	3	19
E	X	X	X	X	X	92
F	X	X	X	X	X	X

Figure 2.

An example of the spectral graph created using F-set metric is shown along with the adjacency matrix. The example illustrates that not all of the combinations in the matrix have to be filled up to determine the clusters and a few weighted edges have to be calculated to determine the clusters. In this example it can be seen that only edges (denoted by blue and red) are necessary for successful clustering calculations. Different color edges represent distinct clusters.

```

int comparekmeriset(string first_string, string second_string){
/* splitfunction(string X) takes a string X and changes it into vector where
each entry in vector corresponds to a single F-set */
vector<string>firstv=splitfunction(first_string);
vector<string>secondv=splitfunction(second_string);
int count=0,i=0,j=0;
#pragma omp parallel for reduction(+:count)private(i,j)
for(i=0;i<firstv.size();i++){
    for(j=0;j<secondv.size();j++){
        if(firstv[i].compare(secondv[j])==0){
            count++;
        }
    }
}
return count;
}

```

Figure 3.

Counting the number of *F-sets* common between two given spectra. Vectors *firstv* and *secondv* contains *F-sets* created from the spectra of a specified size

```

int count=0,i=0,j=0;
#pragma omp parallel for shared(kmer_set)private(i,j)
for(i=0;i<TOTAL_ITEMS;++i){
    if(kmer_set[i]!="XX"){
        for(j=i+1;j<TOTAL_ITEMS;++j){
            threshold=30;
            if(kmer_set[i]!="XX" && kmer_set[j]!="XX"){
                if(comparekmer(kmer_set[i],kmer_set[j])>=threshold){
                    kmer_set[j]=mystring();
                }
            }
        }
    }
    kmer_set[i]=mystring();
}

```

Figure 4.

Parallelizing of the main loop used for calculating weights on edges of the Graph G using OPENMP pragmas

Figure 5.

A schematic of exploiting instruction level parallelism using SIMD instruction set from Intel for thread i is shown. For two vectors to be compared, our strategy divides both vectors in subsets of size 4. Then the four elements from vector A can be compared with 4 elements in vector B. In order to make the comparisons complete, the four elements of vector B are shuffled 3 times. Comparisons of first 4 elements of vector A with 4 elements of vector B (total 16 comparisons) can be done simultaneously due to hardware support in Nehalem architecture. This is repeated until all elements of vector A and B are compared. Note that this scheme is shown for a *single* thread in the diagram and multiple threads can be executed in parallel using multicore architectures.

```

int comparekmeriset(string firstkmeriset,string secondkmeriset){
int count=0, count_sse=0,i=0,j=0;

std::vector<std::string> firstv, secondv;
firstv=splitfunction(firstkmeriset,',');
secondv=splitfunction(secondkmeriset,',');

int n_first=firstv.size(),n_second=secondv.size(), n=max(n_first,n_second);

__attribute__((aligned(16))) float a[n_first], b[n_second], c[n_first+n_second], first_mask_array[4],
second_mask_array[4], third_mask_array[4], fourth_mask_array[4];
__m128 rA, rB, rC, first_shuffle_register, second_shuffle_register, third_shuffle_register,
first_cmp_register, second_cmp_register, third_cmp_register, fourth_cmp_register;

first_shuffle_register=_mm_setzero_ps();second_shuffle_register=_mm_setzero_ps();
third_shuffle_register=_mm_setzero_ps();
rA=_mm_setzero_ps();rB=_mm_setzero_ps();rC=_mm_setzero_ps();
first_cmp_register=_mm_setzero_ps();second_cmp_register=_mm_setzero_ps();
third_cmp_register=_mm_setzero_ps();fourth_cmp_register=_mm_setzero_ps();

for (i = 0; i<n_first ; i+=4) {
rA = _mm_load_ps(&a[i]);
for(j=0;j<n_second;j+=4){
rB = _mm_load_ps(&b[j]);

first_shuffle_register = _mm_shuffle_ps(rB,rB,_MM_SHUFFLE(0,3,2,1));
second_shuffle_register =
_mm_shuffle_ps(first_shuffle_register,first_shuffle_register,_MM_SHUFFLE(0,3,2,1));
third_shuffle_register=
_mm_shuffle_ps(second_shuffle_register,second_shuffle_register,_MM_SHUFFLE(0,3,2,1));

first_cmp_register=_mm_cmpneq_ps(rA,rB);
_mm_store_ps(&first_mask_array[0],first_cmp_register);
second_cmp_register=_mm_cmpneq_ps(rA,first_shuffle_register);
_mm_store_ps(&second_mask_array[0],second_cmp_register);
third_cmp_register=_mm_cmpneq_ps(rA,second_shuffle_register);
_mm_store_ps(&third_mask_array[0],third_cmp_register);
fourth_cmp_register=_mm_cmpneq_ps(rA,third_shuffle_register);
_mm_store_ps(&fourth_mask_array[0],fourth_cmp_register);

for(int y=0;y<4;y++){
if(first_mask_array[y]==0)count_sse++;
if(second_mask_array[y]==0)count_sse++;
if(third_mask_array[y]==0)count_sse++;
if(fourth_mask_array[y]==0)count_sse++;
}
}
}
return count_sse;
}

```

Figure 6.

Counting the number of *F-sets* common between two given spectra using SIMD instructions within each thread. Vectors *firstv* and *secondv* contains *F-sets* created from the spectra of a specified size. This method is called by the main loop which is threaded and each instance of this method is ran by an independent thread.

Figure 7.

The upper array is the default spectra input that will produce very unbalanced loads. As shown in the example figure for 3 threads, the first thread or core would have the greatest load, and this load would not be reduced for other cores i.e. the elimination of the possibilities is not 'propagated' across cores leading to very unbalanced loads. In the redistributed load (shown in the lower array), it can be seen that the computation done on one of the cores would lead to reduced load on other cores. Hence the load is more evenly distributed over the cores leading to more balanced computations across cores. Since, each of the spectra (node) is scheduled to be compared with each of the other spectra (node), the redistribution allows the algorithm to reduce the possibilities that need to be compared on other cores. Although, the redistribution and updating on the global memory incurs a communication cost, it is much less than the cost to actually compare the spectra.

Figure 8.

Wall Clock time with different scheduling strategies for 2000 and 4000 randomly selected spectra is shown. The numbers on the line graphs show the *chunk* size used for *static* scheduling. The other scheduling strategies investigated were *dynamic*, *dynamic & ordered* and *ordered*. Considering the intelligent matrix procedure, the *static* scheduling strategy works best as discussed in the load-balancing section. The *chunk* size seems to have minimal effect on the strategy.

Figure 9.

CPU time and the speedups with increasing size of the data set and increasing number of threads. Number on the lines in the graphs represent the size of the data set

Figure 10.

CPU time and the speedups with increasing size of the data set and increasing number of threads with instruction-level parallel constructs of SSE instructions. Number on the lines in the graphs represent the size of the data set

Figure 11.

CPU time and the speedups with increasing size of the data set, increasing number of F-set size and increasing number of threads with instruction-level parallel constructs of SSE instructions.

Table 1

Summary of results for HCD and CID datasets. The average size of the cluster and the average weighted accuracy of the datasets are shown.

Compression(%) is equal to $\frac{Initialsize - newcompressedsize}{Initialsize} \times 100$

CID datasets	No. of spectra	No. of clustered spectra	No. of clusters	AWA(%)	Compression (%)
AQP2-H-(S256/S261)	2816	593	174	97.3	14.9
AQP2-M-(S256/S261)	2553	662	171	98.1	19.3
AQP2-L-(S256/S261)	2386	531	167	98.5	15.3
AQP2-H-(S264)	1842	485	153	98.7	18.0
AQP2-M-(S264)	1919	499	167	98.8	17.3
AQP2-L-(S264)	2163	490	166	96.3	15.0
HCD data sets					
AQP2-H-(S256/S261)	2282	254	76	99.2	7.8
AQP2-M-(S256/S261)	2713	262	72	99.2	7.0
AQP2-L-(S256/S261)	2754	283	84	98	7.2
AQP2-H-(S264)	2060	132	44	98.5	4.3
AQP2-M-(S264)	2084	155	54	100	4.8
AQP2-L-(S264)	2267	164	57	100	4.7

The effect of clustering with varying F-set size and threshold ζ on the number of high confidence matches (in %) for UPS data is shown. The parameters illustrated are C-x-y where x=F-set size and $y=\zeta$ and UPS-z shows the UPS data with z molarity

Table 2

UPS Dataset	C-5-30	C-10-30	C-20-30	C-30-30	C-40-30
UPS-200	29.9	49.8	74.4	80.3	78.6
UPS-50	32.9	54.7	75.2	81.1	79.2
UPS-10	37.1	59.9	80.1	82.2	80.2