CHAPTER

ACCELERATED STRUCTURAL BIOINFORMATICS FOR DRUG DISCOVERY

5

Wei P. Feinstein, Michal Brylinski
Louisiana State University, USA

Computer-aided design is one of the critical components of modern drug discovery. Drug development is routinely streamlined using computational approaches to improve hit identification and lead selection, to enhance bioavailability, and to reduce toxicity. In the last decade, a mounting body of genomic knowledge has been accumulated due to advancements in genome-sequencing technologies, presenting great opportunities for pharmaceutical research. However, new challenges also arose because processing this large volume of data demands unprecedented computing resources. On the other hand, the state-of-the-art heterogeneous systems currently deliver petaflops of peak performance to accelerate scientific discovery. In this chapter, we describe the development and benchmarking of a parallel version of *e*FindSite, a structural bioinformatics algorithm for the identification of drug-binding sites in proteins and molecular fingerprint-based virtual screening. Thorough code profiling reveals that structure alignment calculations in *e*FindSite consume approximately 90% of the wall-clock time. Parallelizing this portion of the code using pragma-based OpenMP enables the desired performance improvements, scaling well with the number of computing cores.

Compared to a serial version, the parallel code runs 11.8 and 10.1 times faster on the processor and the coprocessor, respectively; when both resources are utilized simultaneously, the speedup is 17.6. By comparing the serial and parallel versions of *e*FindSite, we show the OpenMP implementation of structure alignments for many-core devices. With minimal modifications, a complex, hybrid C++/Fortran77 code was successfully ported to a heterogeneous architecture yielding significant speedups. This demonstrates how modern drug discovery can be accelerated by parallel systems equipped with Intel® Xeon Phi™ coprocessors.

In this chapter, we show solutions to challenges in moving this code to parallelism that are lessons with wide applicability. For instance, we tackle porting extensive use of thread-unsafe common blocks in the Fortran77 code using OpenMP to make thread-private copies. We also enlarged stack sizes to avoid segmentation fault errors [`ulimit -s`]. Serial and parallel versions of *e*FindSite are freely available; please see "For more information" at the end of this chapter.

PARALLELISM ENABLES PROTEOME-SCALE STRUCTURAL BIOINFORMATICS

Advances in genome-sequencing technologies gave rise to the rapid accumulation of raw genomic data. Currently, one of the biggest challenges is to efficiently annotate this massive volume of biological sequences. Due to the prohibitively high costs associated with large-scale experiments, the most practical strategy is computation-based protein structure modeling followed by function inference, also known as structural bioinformatics. This approach routinely produces functional knowledge facilitating a wide range of research in biological sciences; for instance, cellular mechanisms can be investigated by constructing complex networks of molecular interactions at the level of complete proteomes. Systems-level research provides useful insights to support the development of new treatments for complex diseases, which often require a simultaneous targeting of multiple proteins. Consequently, polypharmacology that builds upon systems biology and drug discovery holds a great promise in modern medicine. Incorporating large biological datasets has become one of the central components in systems-level applications; however, significant challenges arise given the vast amount of data awaiting functional annotation. In order to achieve an acceptable time-to-completion in large projects, unprecedented computing power is needed.

In that regard, modern research-driven computer technology is shifting from the traditional single-thread to multiple-thread architectures in order to boost the computing power. Parallel high-performance computing (HPC) has become a key element in solving large-scale computational problems. For example, the Intel® Xeon Phi™ coprocessor featuring Intel® Many Integrated Cores (MIC) architecture offers massively parallel capabilities for a broad range of applications. The underlying x86 architecture supports common parallel programming models providing familiarity and flexibility in porting scientific codes. To take advantage of this unique architecture, we developed a parallel version of *e*FindSite for HPC systems equipped with Intel Xeon Phi coprocessors. *e*FindSite is a template-based modeling tool used in structural bioinformatics and drug discovery to accurately identify ligand-binding sites and binding residues across large datasets of protein targets. In most of the cases, we may expect the sequence similarity between a target protein and a template to be quite low, therefore, *e*FindSite was designed to make reliable predictions using only weakly homologous templates selected from the so-called "twilight zone" of sequence similarity. Consequently, its primary applications are genome-wide protein function annotation, drug design, and systems biology, which demand a sufficient computational throughput as well.

Briefly, *e*FindSite extracts ligand-binding knowledge from evolutionarily related templates stored in the Protein Data Bank (PDB), which are identified using highly sensitive protein threading and meta-threading techniques. Subsequently, ligand-bound templates are structurally aligned onto the target protein in order to detect putative binding pockets and residues. *e*FindSite predictions have a broad range of biological applications, such as molecular function inference, the reconstruction of biological networks and pathways, drug docking, and virtual screening. In the original version of *e*FindSite, template-to-target structure alignments are performed sequentially, thus many processor hours may be required to identify ligand-binding sites in a target protein. The slow modeling process complicates genome-wide applications that typically involve a considerable number of protein targets and large template libraries. In this chapter, we describe porting *e*FindSite to Intel Xeon Phi coprocessors and demonstrate the improved performance of the parallel code in detecting ligand-binding sites across large protein datasets. We show that executing *e*FindSite on computing nodes equipped with coprocessors greatly reduces the simulation time offering a feasible approach for genome-wide protein function annotation, structural bioinformatics, and drug discovery.

OVERVIEW OF *e*FindSite

How do drugs cure diseases? This is a complex question that can be simplified by focusing on the process of protein-ligand binding. In general, small ligand molecules, such as metabolites and drugs, bind to their protein targets at specific sites, often referred to as binding pockets. Ligand binding to proteins induces biological responses either as normal cellular functions or as therapeutic effects to restore homeostasis. On that account, the study of protein-ligand binding is of paramount importance in drug discovery. Since only ligand-free experimental structures and computationally constructed models are available for many pharmacologically relevant proteins, the detection of possible ligand-binding sites is typically the first step to infer and subsequently modulate their molecular functions. Currently, evolution/structure-based approaches for ligand-binding prediction are the most accurate and, consequently, the most widely used. Here, structural information extracted from evolutionarily weakly related proteins, called templates, is exhaustively exploited in order to identify ligand-binding sites and binding residues in target proteins that are linked to certain disease states. One example of an evolution/structure-based approach is *e*FindSite, a ligand-binding site prediction tool. It employs various state-of-the-art algorithms, such as meta-threading by *e*Thread to collect template proteins, clustering by Affinity Propagation to extract ligand information, and machine learning to further increase the accuracy of pocket detection. An important feature of *e*FindSite is an improved tolerance to structural imperfections in protein models, thus it is well suited to annotate large proteomic datasets. For a given target protein, *e*FindSite first conducts structural alignments between the target and each template from the ligand-bound template library. Next, it clusters the aligned structures into a set of discrete groups, which are subsequently ranked in order to predict putative ligand-binding pockets.

BENCHMARKING DATASET

All simulations in this study are performed using a benchmark dataset of protein-ligand complexes comprising proteins of different sizes and a varying number of associated ligand-bound templates. These proteins were selected from the original set of 3659 complexes used previously to develop and parameterize *e*FindSite to mimic a typical proteomic dataset. As shown in Figure 5.1, we first

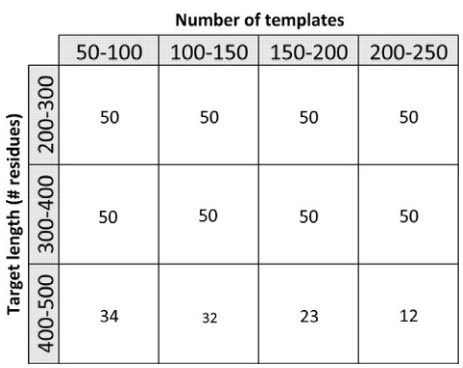

FIGURE 5.1

Benchmarking dataset of 501 proteins.

defined 12 bins with an increasing number of amino acids in target proteins ranging from 200 to 500 as well as the number of associated templates varying from 50 to 250. Next, we randomly selected up to 50 proteins to populate each bin. The benchmarking dataset used in this project contains 501 proteins. For each protein, we use its experimental conformation as the target structure and weakly homologous templates identified by *e*Thread sharing less than 40% sequence identity with the target.

CODE PROFILING

Before converting the serial version of *e*FindSite to a parallel implementation, we conducted a thorough code profiling in order to figure out which portions of the code consume the most CPU cycles. According to the Pareto principle, also known as the 80/20 rule, most computer programs spend 80% of the wall time executing only 20% of the code. On that account, the single most important step prior to porting *e*FindSite to Intel Xeon Phi coprocessors was to identify a block of code that uses the majority of computing time. To conduct code profiling, we randomly selected one protein from each bin presented in Figure 5.1. Figure 5.2 shows the profiling outcome using gprof, a widely used performance analysis tool generating a function list based on the execution time. Four functions were identified as the most time-consuming: tmsearch, cal_tmscore, dp, and get_score taking 29%, 27%, 21%, and 15% of the entire execution time, respectively. Importantly, all these functions contribute to the calculation of structure alignments utilizing up to 92% of the computing time. Next, we measured the wall time by dividing *e*FindSite into three major steps, pre-alignment calculations, structure alignments, and post-alignment processing. As shown in Figure 5.2, structure alignments take 88% of the simulation time, which is consistent with the timings reported by gprof. These profiling results clearly point out at the template-to-target structure alignments as the most computationally expensive operations, thus parallelizing that this portion of the *e*FindSite code should result in the most cost-effective performance improvement in predicting ligand-binding sites.

In principle, there are three approaches to port serial codes to the coprocessor, native, offload, and symmetric modes. In the native mode, the entire code is first cross-compiled on the processor with the parallel regions marked by OpenMP pragmas, and then it is executed directly on the coprocessor. In contrast, the offload mechanism moves only portions of the code to the coprocessor for parallel

Method	Main functions	Time (%)
Gprof	tmsearch	29
	cal_tmscore	27
	dp	21
	get_score	15
Wall time	Pre-alignment	11
	Structure alignment	88

FIGURE 5.2

Profiling of *e*FindSite. Low-level function usage reported by gprof and the wall-clock time measured for the individual stages of *e*FindSite.

computation while leaving the rest of the serial code running on the processor. In the symmetric implementation, coprocessors are used independently as self-sufficient computing nodes often through an MPI protocol. We looked into the source code to determine which parallel mode would be optimal for *e*FindSite. We gave careful thought to several considerations. The framework of *e*FindSite is written in C++, whereas protein structure alignments are implemented in Fortran77. For each individual template-to-target structure alignment, the subroutine frtmalign is called from the main function. Figure 5.3 shows the detailed call graph generated by Doxygen, illustrating nested function calls by frtmalign and its subroutines. Transferring structure alignment computations to the coprocessor using Fortran-specific OpenMP pragmas would require the minimum code conversion. In addition, several pre- and post-alignment functions such as the Affinity Propagation clustering used to group template-bound ligands are available only as external libraries precompiled for Linux hosts. Moreover, according to the code profiling carried out previously, the total memory footprint of *e*FindSite grows with the target protein size and the number of associated templates to values that are prohibitively large considering the memory space available on the coprocessor. Therefore, we ruled out the native execution and decided to "offload" only the structure alignment portion to the coprocessor leaving the rest

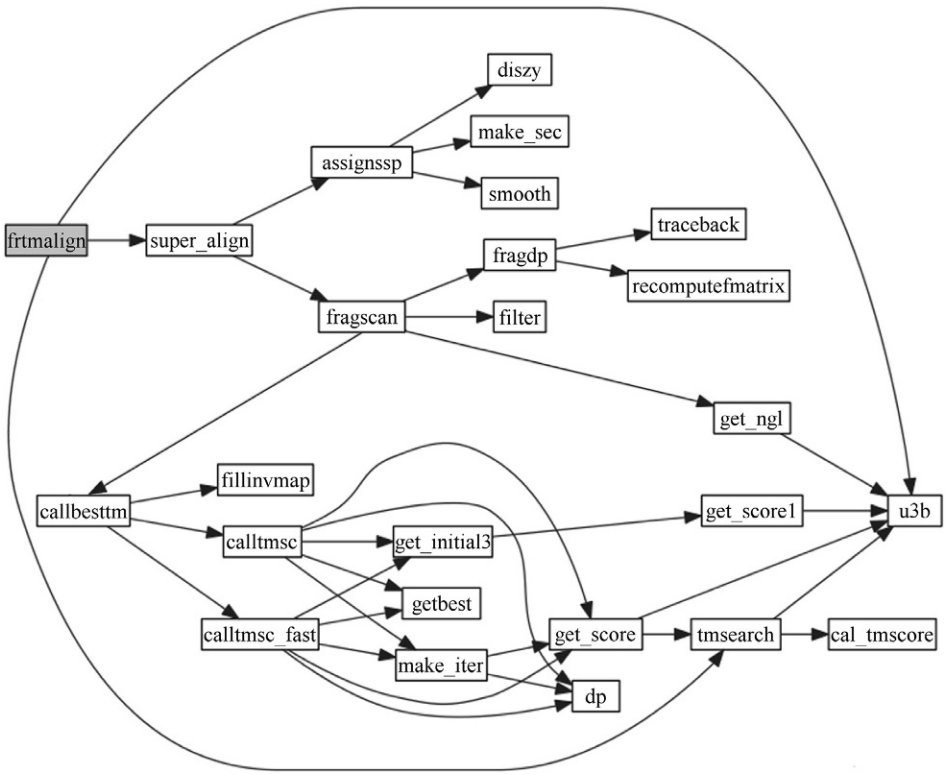

FIGURE 5.3

Call graph of subroutines for protein structure alignments. The top subroutine of frtmalign is called within the main function of *e*FindSite.

of the code, including pre- and post-alignment calculations, on the processor. In this chapter, we discuss the parallelization of *e*FindSite using the offload mode followed by the performance analysis against 501 target proteins.

PORTING *e*FindSite FOR COPROCESSOR OFFLOAD

The flowchart of the parallel version of *e*FindSite is shown in Figure 5.4. *e*FindSite takes a target protein structure and a ligand-bound template library as an input to identify ligand-binding sites. As a result, it produces a series of informative predictions, including putative ligand-binding sites and binding residues, structure alignments between the target and template proteins, and ligand molecular fingerprints for virtual screening. The workflow central to the binding site prediction breaks down into three stages. During the pre-alignment step, *e*FindSite extracts template information from the library and constructs global sequence alignments to the target; these computations are performed sequentially on the processor. Next, it structurally aligns each template onto the target; this section is well suited for parallel computing since individual alignments are fully independent of each other. The code profiling also identified this portion of the code as the most computing intense. Therefore, we decided to parallelize the structure alignment loop in order to perform the calculations using

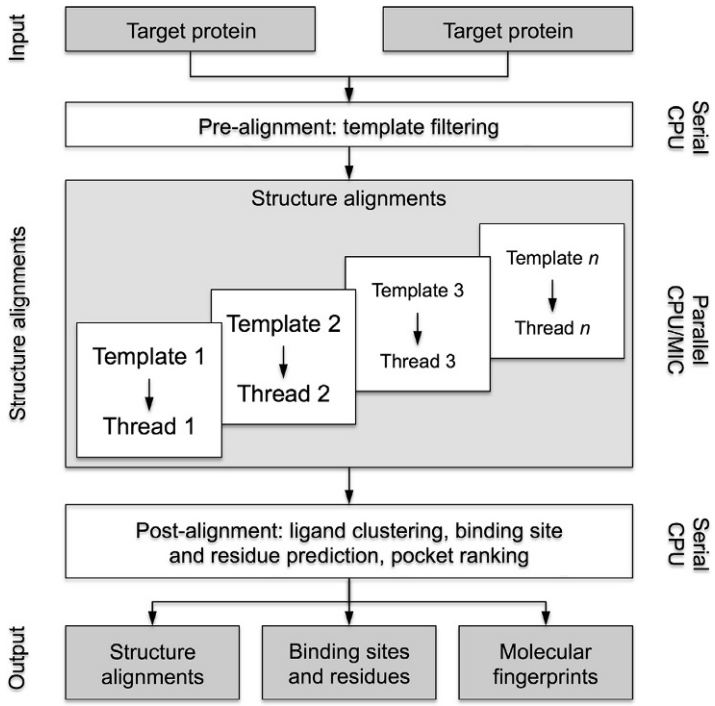

FIGURE 5.4

Workflow of *e*FindSite. Pre- and post-alignment calculations are performed using a single processor thread. Structure alignments are executed in parallel using multiple threads on the processor and/or coprocessor.

multiple processor and/or coprocessor threads. Figure 5.4 shows that each structure alignment is assigned to a unique hardware thread, so that different threads concurrently carry out calculations for different templates against the target protein. Once structure alignments are completed, the results are sent back to the processor, which clusters and ranks the identified binding pockets, and predicts the corresponding binding residues in the final post-alignment step.

In the following sections, we discuss some details of porting *e*FindSite to the coprocessor. Within the main function, structure alignments are executed inside a master loop iterating over the template library. The parallelization of structure alignments is implemented by executing each alignment concurrently using standard OpenMP pragmas as shown in Figure 5.5. The execution of a structure alignment starts when the wrapper function frtmalign is called within the main function, where the serial subroutine frtmalign implemented in Fortran77 is invoked to conduct the actual alignment calculation. We found that this portion of the code extensively uses common blocks making structure alignments thread-unsafe. To guarantee the thread safety, we marked all Fortran77 common blocks as private to threads using [`!$omp threadprivate (/block_name/)`], which is shown in Figure 5.6.

This implementation of *e*FindSite offloads portions of the code to the coprocessor, executes structure alignments on the coprocessor, and then transfers the results back to the processor to perform the final post-alignment calculations. Because the coprocessor has its own memory, the offload mode requires data transfer between the processor and coprocessor. *e*FindSite implements C++ classes to store and manipulate data associated with template proteins, such as atomic coordinates, amino acid residues, and sequences. To facilitate data transfer, we devoted extra work converting the original data structures into flat, bitwise copy-able data arrays before transferring them to the coprocessor as illustrated in Figure 5.7. We note that arrays containing the target protein and the template library are copied to the coprocessor only once, thus there is virtually no overhead from moving data back and forth. Each structure alignment

```
//loop through the library of template proteins to conduct structure alignments
std::multimap< int, Template *, greater<int> >::iterator tpl1;
//parallelize the loop
#pragma omp parallel for schedule(dynamic) private(tpl1)
for (tpl1 = template_set.begin(); tpl1 != template_set.end(); tpl1++ ){
        char template_seq[MAXPRO];
        int template_res[MAXPRO];
        double template_xyz[MAXPRO][3];
//assemble protein residue, sequence and coordinate from the template library
        int template_len = ((*tpl1).second)->getProteinResiduesTotal();
        strcpy(template_seq, (((*tpl1).second)->getProteinSequence()).c_str());
        for ( int t_i = 0; t_i < template_len; t_i++ ){
                template_res[t_i] = t_i + 1;
        }
        ((*tpl1).second)->getProteinCoordsCA(template_xyz);
        int t_score1, t_alig[MAXPRO];
        double t_score2, t_score3, t_score4, t_t[3], t_u[3][3];

// call structure alignment subroutine
        frtmalign_(&t_score1, &t_score2, &t_score3, &t_score4, &template_len,\
        &target_len, &template_seq, &target_seq, &t_alig, &template_res,\
        &target_res, &template_xyz, &target_xyz, &t_t, &t_u, &align_len);
}
```

FIGURE 5.5

OpenMP parallelization of the master loop performing structure alignments between the target protein and a set of ligand-bound templates.

```
      subroutine
      frtmalign(tm_score1,tm_score2,tm_score3,tm_score4,target_len,template_len,
     &        target_seq,template_seq,tm_align,target_res,template_res,
     &        target_xyz,template_xyz,tm_tt1,tm_uu1,align_length)

      parameter (maxres=1000)           ! no. of residues
      parameter (maxlen=2*maxres)       ! for alignment
      parameter (npr=2)                 ! no. of proteins to align
      double precision tm_score2,tm_score3,tm_score4,tm_xyz1a(0:2,0:maxres-1),
     &    template_xyz(0:2,0:maxres-1), tm_tt1(0:2),tm_uu1(0:2,0:2)
      integer tm_score1,target_len,template_len, align_length,
     &    tm_align(0:maxres-1), target_res(0:maxres-1),
     &    template_res(0:maxres-1)
      character*1 target_seq(0:maxres-1), template_seq(0:maxres-1)
      character*3 resa(maxres,npr),resn(maxres,0:1)
      character*1 seqa(maxres,npr),seqn(maxres,0:1)
      dimension invmap0(maxres),ires(maxres,0:1)
      dimension xtm1(maxres),ytm1(maxres),ztm1(maxres)
      dimension xtm2(maxres),ytm2(maxres),ztm2(maxres)
      dimension m1(maxres),m2(maxres)
      dimension xyz(3,maxres,npr),length(npr)
      common /coord/ xa(3,maxres,0:1)
      common /length/ nseq1,nseq2
      common /pdbinfo/ ires1(maxres,npr),resa,seqa
      common /secstr/ isec(maxres),jsec(maxres)     !secondary structure
      common /n1n2/ n1(maxres),n2(maxres)
      common /dinfo/ d8
      common /stru/xt(maxres),yt(maxres),zt(maxres),xb(maxres),yb(maxres),
     &       zb(maxres)
      // OpenMP pragma to mark thread private
        !$omp threadprivate(/coord/)
        !$omp threadprivate(/length/)
        !$omp threadprivate(/pdbinfo/)
        !$omp threadprivate(/secstr/)
        !$omp threadprivate(/n1n2/)
        !$omp threadprivate(/dinfo/)
        !$omp threadprivate(/stru/)

cc    call other routines...

      end
```

FIGURE 5.6

Thread-safe implementation of common blocks in Fortran77.

executed on the coprocessor extracts data required for a given template from the transferred data block using an offset mechanism. In addition, we use compiler directives to mark the offload execution, as presented in Figure 5.8. Specifically, all Fortran77 subroutines and global variables for structure alignment calculations on the coprocessor are tagged with the offload attributes [`!dir$ attributes offload:mic::subroutine_name`] and [`!dir$ attributes offload:mic::variable_name`], respectively. Moreover, when the OpenMP is invoked within the offloaded block, environment variables for the coprocessor are set using the "MIC_" prefix, i.e., [`export MIC_OMP_NUM_THREADS=n`]. Since the memory footprint for individual structure alignments in *e*FindSite is larger than the default OpenMP stack size, we increase it by using [`export MIC_OMP_STACKSIZE=64M`].

The Intel Xeon Phi coprocessor model SE10P features 61 cores and supports up to 240 parallel threads with the 61st core reserved for operating system, I/O operations, etc., when using offload. The physical distribution of threads over hardware cores is controlled by the thread affinity settings,

```
 int i_offset = 0;
 // flatten the data structure to data array
 for ( tpl1 = template_set.begin(); tpl1 != template_set.end(); tpl1++ ){
        template_len[i_offset] = ((*tpl1).second)->getProteinResiduesTotal();
        char t_seqt[MAXPRO];
        strcpy(t_seqt, (((*tpl1).second)->getProteinSequence()).c_str());
        for (int t_i=0;t_i<((*tpl1).second)->getProteinResiduesTotal();
               t_i++ ){
               template_seq[t_i+t_offset[i_offset]] = t_seqt[t_i];
        }
       for (int t_i=0;t_i<((*tpl1).second)->getProteinResiduesTotal();t_i++){
              template_res[t_i+t_offset[i_offset]] = t_i + 1;
       }
       double *t_xyzt=new double [((*tpl1).second)->getProteinResiduesTotal()
       *3];
       ((*tpl1).second)->getProteinCoords1D(t_xyzt);
       for (int t_i 0;t_i<((*tpl1).second)->getProteinResiduesTotal()*3;t_i++)
       {
              template_xyz[t_i+t_offset[i_offset]*3] = t_xyzt[t_i];
       }
       delete [] t_xyzt;
       i_offset++;
}
// offload to the coprocessor, data transfer
#pragma offload target(mic) in(n_offset,t_offset,t_len1) \
                            in(template_len:length(n_set)) \
                            in(target_seq,target_res:length(n_tar)) \
                            in(template_seq,template_res:length(n_off)) \
                            in(target_xyz:length(n_tar*3)) \
                            in(template_xyz:length(n_off*3)) \
                            out(t_score1,t_score2,t_score3,\
                                t_score4:length(n_set)) \
                            out(t_alig:length(n_tar*n_set)) \
                            out(t_rmat:length(n_set*12)){
int tm_i;
// parallelize the for loop
#pragma omp parallel for schedule(dynamic) private(tm_i)
for ( tm_i = 0; tm_i < n_offset; tm_i++ ){
       int o_score1;
       double o_score2, o_score3, o_score4;
       int target_o_len = target_len;
       int template_o_len = template_len[tm_i];
       char target_o_seq[MAXPRO];
       char template_o_seq[MAXPRO];
       int target_o_res[MAXPRO], template_o_res[MAXPRO];
       double target_o_xyz[MAXPRO][3], template_o_xyz[MAXPRO][3];
       for ( int t_i = 0; t_i < target_o_len; t_i++ ){
           target_o_seq[t_i] = target_seq[t_i];
           target_o_res[t_i] = target_res[t_i];
            for ( int t_j = 0; t_j < 3; t_j++ ){
               target_o_xyz[t_i][t_j] = target_xyz[t_i*3+t_j];
             }
       }
       for (int t_i = 0; t_i < o_len2; t_i++ ){
          template_o_seq[t_i] = template_seq[t_offset[tm_i]+t_i];
           template_o_res[t_i] = template_res[t_offset[tm_i]+t_i];
           for ( int t_j = 0; t_j < 3; t_j++ ){
               mplate_o_xyz[t_i][t_j]=template_xyz[t_offset[tm_i]*3+t_i*3+t_j];
           }
    }
   int o_alig[MAXPRO];
   double o_t[3], o_u[3][3];

   frtmalign_(&o_score1, &o_score2, &o_score3, &o_score4, &tempalte_o_len, \
              &target_o_len, &template_o_seq, &target_o_seq, &o_alig, \
              &template_o_res, &target_o_res, &template_o_xyz, \
              &target_o_xyz, &o_t, &o_u, &align_o_len );
}
//end of the parallel for loop
```

FIGURE 5.7

Data flattening and transfer for offloading structure alignments to the coprocessor.

```
c  Fr-TM-align converted into a subroutine

!dir$ attributes offload:mic::frtmalign

!dir$ attributes offload:mic::d8
!dir$ attributes offload:mic::xa
!dir$ attributes offload:mic::nseq1
!dir$ attributes offload:mic::nseq2
!dir$ attributes offload:mic::ires1
!dir$ attributes offload:mic::resa
!dir$ attributes offload:mic::seqa
!dir$ attributes offload:mic::isec
!dir$ attributes offload:mic::jsec
!dir$ attributes offload:mic::n1
!dir$ attributes offload:mic::n2
!dir$ attributes offload:mic::xt
!dir$ attributes offload:mic::yt
!dir$ attributes offload:mic::zt
!dir$ attributes offload:mic::xb
!dir$ attributes offload:mic::yb
!dir$ attributes offload:mic::zb

subroutine frtmalign(tm_score1,tm_score2,tm_score3,tm_score4,target_len,
     &    template_len,target_seq,template_seq,tm_align,target_res,
     &    template_res,target_xyz,template_xyz,tm_tt1,tm_uu1,align_length)
     .
     .
end
```

FIGURE 5.8

Fortran77 subroutine frtmalign offloaded to the coprocessor.

which are typically set to either compact, balanced, or scatter modes. Since the affinity mode may have an impact on the parallel performance, we first benchmarked *e*FindSite using different Intel KMP affinity settings, [`export MIC_KMP_AFFINITY=compact/scatter/balanced`].

In order to analyze the performance of *e*FindSite, we first developed a reliable measure to quantify its computing speed. As shown in Figure 5.3, the subroutine u3b is a frequently called low-level function that calculates a root-mean-square deviation (RMSD) between two sets of atomic coordinates. We found that the total number of RMSD calculations correlates well with the total wall time and the structure alignment time; the corresponding Pearson correlation coefficients are as high as 0.963 and 0.960, respectively. Thus, we use the number of RMSD calculations per second to measure the *e*FindSite performance.

In Figure 5.9, we plot the performance of *e*FindSite with the structure alignment portion of the code offloaded to the coprocessor. Note that the pre- and post-alignment steps are sequentially executed on the processor. Because some proteins in the benchmark dataset have less than 50 templates, we only consider parallel execution using up to 24 threads per task in order to avoid idle threads. Encouragingly, increasing the number of threads on the coprocessor clearly improves the performance of *e*FindSite. The average performance is 4.27×10^4 and 2.30×10^5 RMSD calculations per second as the number of threads increases from 4 to 24. Considering structure alignment calculations alone, almost a perfect linear scaling is achieved (open circles in Figure 5.9). In contrast, the performance of *e*FindSite reaches a plateau when the total wall time is plotted (black circles in Figure 5.9). Here, the number of RMSD calculations per second improves from 3.98×10^4 for 4 threads to 1.76×10^5 for 24 threads, which can be explained by Amdahl's Law describing the relationship between the expected speedup of a parallel implementation relative to the serial algorithm. These performance measurements are carried out for the balanced and scatter thread affinity modes that maximize hardware utilization by evenly spreading

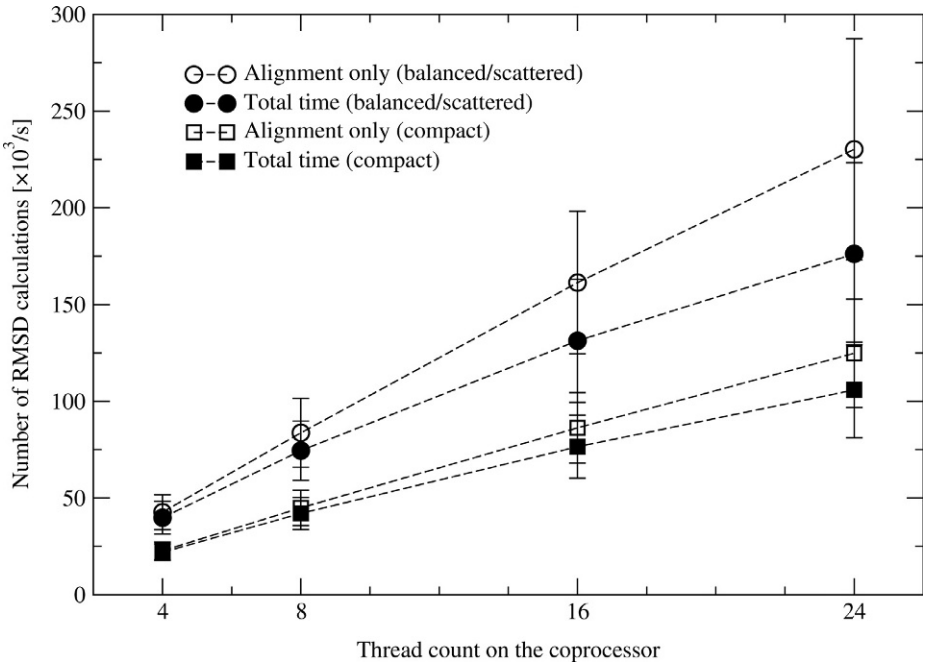

FIGURE 5.9

Performance of *e*FindSite using parallel coprocessor threads. The speed is measured by the number of RMSD calculations per second (mean ± standard deviation) for 4, 8, 16, and 24 threads across the benchmarking dataset of 501 proteins. Solid and open symbols correspond to the total time and the time spent calculating structure alignments, respectively. Circles and squares show the performance using balanced/scatter and compact thread affinity, respectively.

threads across coprocessor cores. For the compact affinity setting placing four threads on a single core before moving to the next one, the rate of RMSD calculations increases from 2.27×10^4 to 1.25×10^5 for the alignment time (open squares in Figure 5.9), and from 2.19×10^4 to 1.06×10^5 for the total time (black squares in Figure 5.9). Although the balanced/scatter thread affinity yields an approximately 1.8 times higher performance per thread compared to the compact affinity mode, its advantage starts diminishing when the total number of threads on the coprocessor exceeds 120, and all affinity modes become essentially the same when the thread count reaches 240. In fact, a core-to-core performance comparison shows a slightly higher performance of the compact mode at 2.44×10^4 RMSD calculations per second compared to 2.38×10^4 for the balanced and scatter modes.

PARALLEL VERSION FOR A MULTICORE PROCESSOR

In addition to the Intel Xeon Phi coprocessor version of *e*FindSite, we also implemented a parallel code that can be executed on multicore processors. Similarly, we employed pragma-based OpenMP to parallelize structure alignment calculations with both pre- and post-alignment steps executed sequentially. We also increased the memory available to each thread to 64M using [`export`

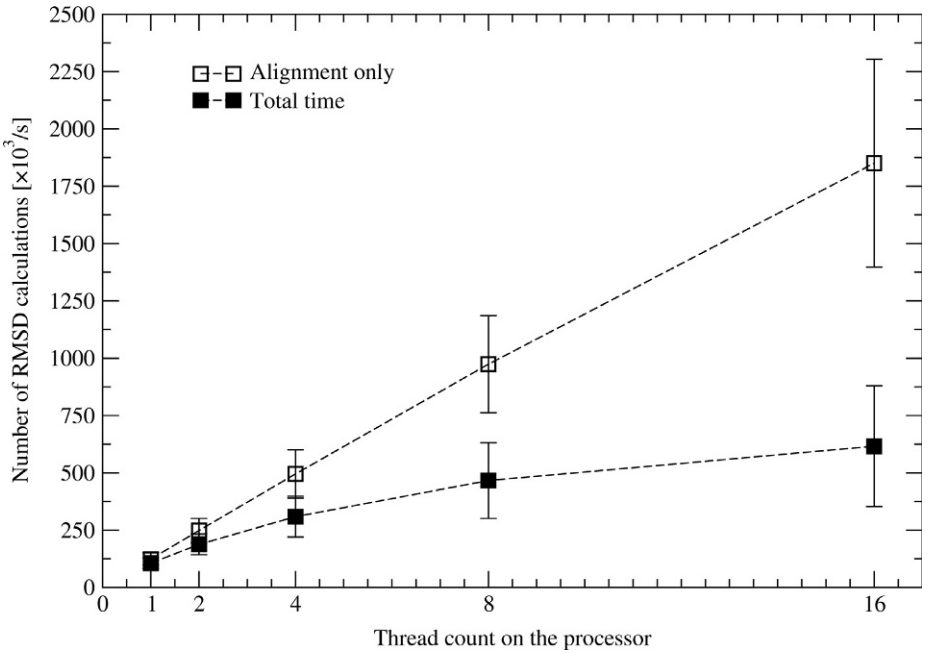

FIGURE 5.10

Performance of *e*FindSite using parallel processor threads. The speed is measured by the number of RMSD calculations per second (mean ± standard deviation) for 2, 4, 8, and 16 threads across the benchmarking dataset of 501 proteins. Solid triangles and open squares correspond to the total time and the time spent calculating structure alignments, respectively.

`OMP_STACKSIZE=64M`]. In Figure 5.10, we plot the performance of the multi-threaded version of *e*FindSite on the Intel® Xeon® E5-2680 processor. Using the total simulation time, the serial (1 thread) and parallel (16 threads) versions have an average performance across the benchmarking dataset of 1.06×10^5 and 6.16×10^5 RMSD calculations per second, respectively. When considering the time spent calculating structure alignments, the average performance increases to 1.24×10^5 and 1.85×10^6 RMSD calculations per second. A good linear scaling is achieved with a speedup over the serial execution of 15 when 16 processor threads are utilized. The performance reaches a plateau at a speedup of 5.8 for the total wall time; we note that according to Amdahl's law, the maximum theoretical speedup of a code that is 90% parallelized using 16 threads is 6.4.

TASK-LEVEL SCHEDULING FOR PROCESSOR AND COPROCESSOR

Computing nodes equipped with Intel Xeon Phi coprocessor cards provide massively parallel capabilities through multicore processors and many-core coprocessors. Therefore, our ultimate goal was to develop a production code of *e*FindSite that takes full advantage of both resources. The production scheme

for predicting ligand-binding sites across large datasets of proteins using *e*FindSite involves using both the processor and the coprocessor simultaneously. The processor executes the serial portion of the code, while structure alignments are offloaded to the coprocessor with the compact affinity mode to maximize the performance of *e*FindSite. At the same time, the processor also runs a parallel version of *e*FindSite as well, using relatively few threads. This way, we process multiple proteins concurrently by multiple parallel tasks executed simultaneously on both computing units.

We developed a job scheduler in Perl to launch up to 4 parallel tasks on the processor with 4 threads per task, and up to 10 parallel tasks on the coprocessor, each using 24 threads in the compact affinity mode. As shown in Figure 5.11, we first sort target proteins by a product of the residue count and the number of templates, which estimates the wall time required to complete the calculations. Because we

```
use Proc::Background;
my $n_mic = 10;        // 10 parallel jobs on a coprocessor
my $n_cpu = 4;         // four parallel jobs on a processor
my @lst3 = ();         // template protein list
foreach my $wlst2 ( sort { $lst2{$a} <=> $lst2{$b} } keys %lst2 ){
        push(@lst3, $wlst2);
}
while ( @lst3 ){
        # dispatching jobs to Xeon processor
        for ( my $xa = 0; $xa < $n_cpu; $xa++ ){
                # start a new process if the current is finished
                if ( @lst3 and !$j_cpu[$xa]->alive ){
                        #get an item from the top of the list (smaller template)
                        my $job = shift(@lst3);
                        printf("CPU%d <- %s ... %.1f%s\n", $xa, $job, \
                                (++$prg1/$nlst3)*100.0, '%');
                        $j_cpu[$xa]=Proc::Background->new("$ef_omp -s $job.pdb \
                          -t $job.ethread-fun -i $job.ss -e $job.prf -o $job- \
                          efindsite -b $foptb -x $foptx > $job.log 2>&1");
                        $c_cpu[$xa]++;
                        $l_cpu[$xa] += $lst2{$job};
                }
        }
        # offloading jobs to Xeon MIC jobs
        for ( my $xa = 0; $xa < $n_mic; $xa++ ){
                if ( @lst3 and !$j_mic[$xa]->alive ){
                    #get an item from the bottom of the list (larger template)
                    my $job = pop(@lst3);
                    my $kpt='export \
                                MIC_KMP_PLACE_THREADS=6c,4t,'.int($xa*6).'O';
                    printf("MIC%d <- %s ... %.1f%s\n", $xa, $job, \
                                (++$prg1/$nlst3)*100.0, '%');
                    $j_mic[$xa] = Proc::Background->new("$kpt ; $ef_mic -s \
                        $job.pdb -t $job.ethread-fun -i $job.ss -e $job.prf -o \
                        $job-efindsite -b $foptb -x $foptx > $job.log 2>&1");
                    $c_mic[$xa]++;
                    $l_mic[$xa] += $lst2{$job};
                }
        }
        sleep(2) if ( @lst3 );
}
```

FIGURE 5.11

Part of the task scheduler implemented in Perl dispatching four parallel tasks (4 threads per task) on the processor and 10 parallel tasks (10 threads per task) on the coprocessor.

use 24 threads per task on the coprocessor and 4 threads per task on the processor, we dispatch longer tasks to the coprocessor, whereas shorter jobs are executed on the processor. Running 4 parallel tasks on the processor and 10 parallel tasks on the coprocessor ensures that both resources are fully utilized. To avoid the oversubscription of coprocessor cores, the production code features an explicit mapping of parallel tasks to hardware threads at the compact thread affinity. Specifically, [`export MIC_KMP_PLACE_THREADS=6c,4t,(i*6).'o'`] defines the set of logical units assigned to each task, where `6c` requests six cores, `4t` starts four threads on each core, and `(i*6).'o'` is an offset of six cores between individual task. For example, when `i=0`, the first task runs on cores 0-5 with 4 threads per core totaling 24 threads, the next task sets `i=1` to run on cores 6 through 11, and so forth. This way, the scheduler assigns different cores to different parallel tasks in order to conduct protein structure alignments at the maximum utilization of the accelerator hardware. Figure 5.12 lists partitioning details for the parallel execution for the benchmarking dataset using a computing node equipped with two 8-core Intel Xeon E5-2680 processors (8 threads, hyper-threading disabled) and one 61-core Intel Xeon Phi SE10P coprocessor (240 threads, compact thread affinity).

Hardware	Core range	Number of threads	Percentage of computations
Processor	1-4	4	12.2
	5-6	4	12.5
	7-11	4	12.7
	12-16	4	11.8
Coprocessor	1-6	24	5.1
	7-12	24	5.1
	13-18	24	5.1
	19-24	24	5.0
	25-30	24	5.3
	31-36	24	5.0
	37-42	24	4.7
	43-48	24	5.0
	49-54	24	5.4
	55-60	24	5.1
Processor	1-16	16	49.2
Coprocessor	1-60	240	50.8

FIGURE 5.12

Partitioning details for processing 501 benchmarking proteins using *e*FindSite simultaneously on the processor and coprocessor. The amount of computations is approximated by the product of the target protein length and the number of template structures.

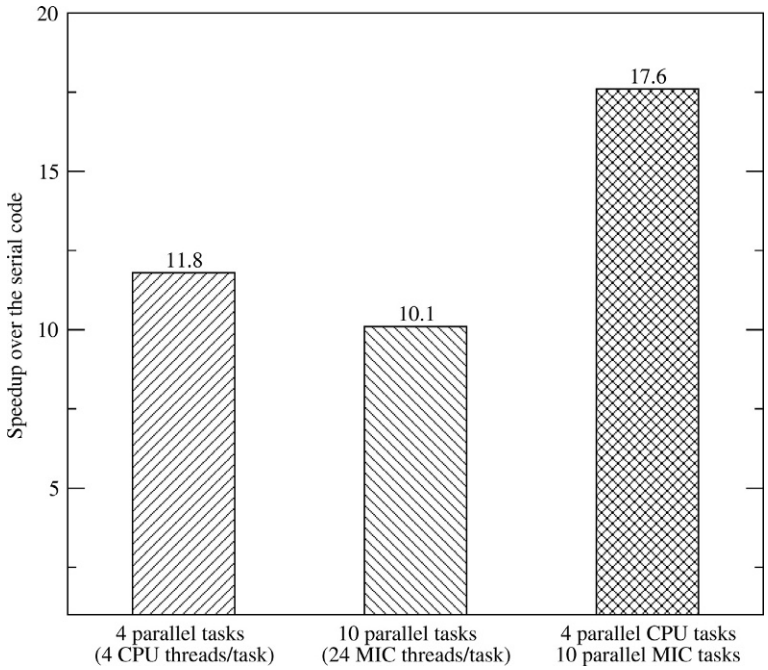

FIGURE 5.13

Speedups of the parallel versions of *e*FindSite over the serial code. Computations are performed using three parallel versions: multiple processor threads (4 tasks, each running on 4 threads), multiple coprocessor threads (10 tasks, each running on 24 threads), and multiple processor and coprocessor threads running simultaneously.

We processed the entire dataset of 501 proteins using task-level parallelism and measured the time-to-completion, defined as the total time required for the prediction of ligand-binding sites. Figure 5.13 compares the results for the serial version and three parallel processing schemes. The speedup over the serial version is 11.8, 10.1, and 17.6 for the parallel processing using the processor, the coprocessor, and both resources, respectively. Therefore, using the coprocessor in addition to the processor accelerates the prediction of binding sites across large protein datasets. We also monitored the utilization of computing resources to demonstrate that all hardware threads are utilized. Figure 5.14 shows that the average usage of the processor and the coprocessor during production simulations is 99.9% and 82.2%, respectively. The coprocessor threads periodically become idle while waiting for the processor to start the pre-alignment (reading input files, performing sequence alignments) and finish the post-alignment (pocket clustering and ranking) tasks resulting a relatively lower utilization of the coprocessor. On the other hand, the processor remains fully utilized not only facilitating tasks offloaded to the coprocessor but also performing *e*FindSite calculations by itself. Finally, we checked for the numerical correctness of the results. Different versions of *e*FindSite produce identical results, thus the parallelization of *e*FindSite using OpenMP and offloading to the Intel Xeon Phi coprocessor fully maintains the functionality of the original code with a great benefit of much shorter simulation times.

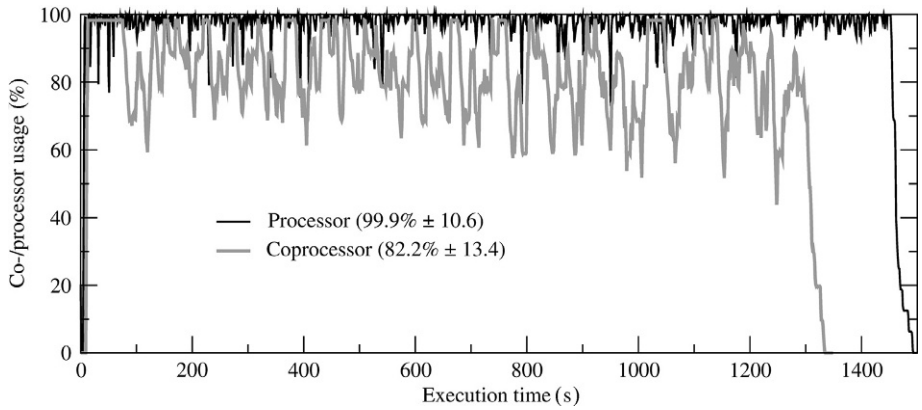

FIGURE 5.14

Resource utilization during the execution of the heterogeneous parallel version of *e*FindSite. Time courses of the percentage of processor usage (black line) are compared to that of the coprocessor (gray line).

CASE STUDY

Finally, we present a case study to illustrate binding pocket prediction using *e*FindSite. The target protein selected from the benchmarking dataset is human arginase I (PDB-ID: 3gn0, chain A), a binuclear manganese metalloenzyme hydrolyzing L-arginine. The abnormal activity of this protein is implicated in various disease states including erectile dysfunction, atherosclerosis, and cerebral malaria. *e*FindSite predicted a total of 10 pockets for this protein and assigned a confidence score of 91.9% to the top-ranked binding site. Figure 5.15 shows the crystal structure of this protein (transparent ribbons) with the top-ranked binding pocket predicted by *e*FindSite marked by a solid ball. The corresponding predicted binding residues are shown as a transparent gray surface. Two additional smaller balls mark the location of pockets at ranks 2 and 3. The prediction accuracy can be evaluated by revealing the location of a ligand α-difluoromethylornithine bound to the target protein in the experimental complex structure represented by solid sticks. We note that the ligand position was not part of the prediction procedure and it is used for validation purposes only. The distance between the predicted top-ranked binding site and the geometric center of the ligand is only 2.22 Å, demonstrating a high prediction accuracy of *e*FindSite. As a reliable tool for ligand-binding prediction, *e*FindSite is well suited for a broad range of applications ranging from protein function annotation to virtual screening and drug discovery.

SUMMARY

*e*FindSite is a ligand-binding site prediction software used in drug discovery and design. To meet the challenges of proteome-scale applications, we implemented a parallel version of *e*FindSite for processing large datasets using HPC systems equipped with Intel Xeon Phi coprocessors. This parallel version of *e*FindSite achieves 17.6 speedup over the serial code using both processor and coprocessor

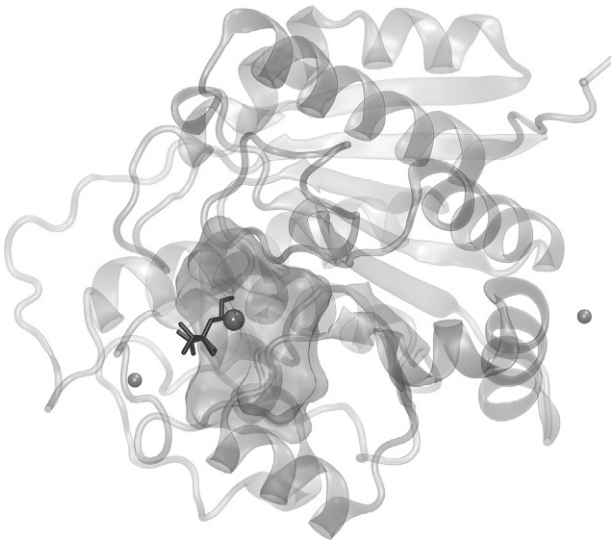

FIGURE 5.15

Ligand-binding pocket prediction for human arginase I using *e*FindSite. The crystal structure of the target protein and the binding ligand is displayed as transparent gray ribbons and solid black sticks, respectively. The top-ranked predicted binding site is shown as a solid ball representing the pocket center and a transparent molecular surface around the binding residues. Two smaller balls represent the centers of binding pockets predicted at ranks 2 and 3.

resources, which is considered significant when normalized by the cost of the hardware. The x86 coprocessor architecture allows for open-standard and portable parallel programming on both traditional processors as well as coprocessors. In particular, using OpenMP makes it relatively easy to parallelize portions of the code. *e*FindSite represents a typical scientific software written mostly by domain scientists, who contributed different components to the code using different programming languages and styles. From our porting experience, even fairly complex codes, such as the hybrid C++/Fortran77 source code of *e*FindSite, can be successfully ported to utilize both Intel Xeon processors and Intel Xeon Phi coprocessors together yielding speedups at minimal coding efforts.

The development of the parallel version of *e*FindSite was fairly straightforward; however, we would like to discuss a couple of problems encountered throughout this process. First, due to the extensive use of thread-unsafe common blocks in the Fortran77 code, we initially attempted to rewrite those subroutines to pass parameters explicitly instead of accessing common blocks. However, these efforts took a very long time and considering the "spaghetti-like" nature of the code with multiple access points to common blocks, we could not reproduce the correct results in a timely fashion. For that reason, we decided to use OpenMP pragmas to mark the global variables as thread-private in the original code. This method turned out to be very practical and time efficient in the process of developing a thread-safe parallel implementation. Another issue we ran into was related to the stack size. We found that the default stack size on some HPC systems is smaller than that set for *e*FindSite using `OMP_STACKSIZE` causing segmentation fault errors. Consequently, in addition to OpenMP pragmas, modifying the default system stack size using [`ulimit -s`] may be required.

Notwithstanding the 17-fold speedup, there is still room for improvements to fully benefit from wide single instruction, multiple data (SIMD) vectors featured by the coprocessor. For instance, a proper loop vectorization is critical to increase the overall performance. Vectorization reports collected for *e*FindSite show that the majority of loops taking the most execution time are indeed vectorized. However, other issues related to data dependency and alignment indicated in the reports need to be addressed by rearranging loops, data structure padding, improving register utilization, and data caching. Therefore, in addition to the parallelization of the remaining portions of the serial code, future directions of this project include a thorough code optimization to take better advantage of the Intel Xeon processor and Intel Xeon Phi coprocessor SIMD architectures. We plan to hand tune a relatively small portion of the kernel for RMSD calculations to further boost the parallel performance of *e*FindSite.

FOR MORE INFORMATION

- The source code for *e*FindSite can be found at www.brylinski.org/efindsite. This Web site also provides compilation and installation instructions, as well as a detailed tutorial on processing large datasets using heterogeneous computing platforms.
- Feinstein, W.P., Moreno, J., Jarrell, M., Brylinski, M., 2015. Accelerating the pace of protein functional annotation with Intel Xeon Phi coprocessors. IEEE Trans. Nanobiosci. DOI: 10.1109/TNB.2015.2403776.
- Brylinski, M., Feinstein, W.P., 2013. *e*FindSite: Improved prediction of ligand binding sites in protein models using meta-threading, machine learning and auxiliary ligands. J. Comput. Aided Mol. Des. 27, 551-567.
- Feinstein, W.P., Brylinski, M., 2014. *e*FindSite: Enhanced fingerprint-based virtual screening against predicted ligand binding sites in protein models. Mol. Inform. 33, 135-150.
- Download the code from this, and other chapters, http://lotsofcores.com.