Code Modernization Tools for Assisting Users in Migrating to Future Generations of Supercomputers

Ritu Arora and Lars Koesterke

Abstract Usually, scientific applications outlive the lifespan of the High Performance Computing (HPC) systems for which they are initially developed. The innovations in the HPC systems' hardware and parallel programming standards drive the modernization of HPC applications so that they continue being performant. While such code modernization efforts may not be challenging for HPC experts and well-funded research groups, many domain-experts and students may find it challenging to adapt their applications for the latest HPC systems due to lack of expertise, time, and funds. The challenges of such domain-experts and students can be mitigated by providing them high-level tools for code modernization and migration. A brief overview of two such high-level tools is presented in this chapter. These tools support the code modernization and migration efforts by assisting users in parallelizing their applications and porting them to HPC systems with high-bandwidth memory. The tools are named as: Interactive Parallelization Tool (IPT) and Interactive Code Adaptation Tool (ICAT). Such high-level tools not only improve the productivity of their users and the performance of the applications but they also improve the utilization of HPC resources.

1 Introduction

High Performance Computing (HPC) systems are constantly evolving to support computational workloads at low cost and power consumption. While the computing density per processor has increased in the last several years, the clock speed of the processors has stopped increasing to prevent unmanageable increase in the temperature of the processor, and to limit the gap between the speed of the processor and the memory. This trend has resulted in HPC systems that are equipped with manycore processors and deep memory hierarchies. To achieve high-performance on such HPC systems, appropriate level of parallelization, vectorization, and memory optimization are critical.

R. Arora (✉) • L. Koesterke
Texas Advanced Computing Center, The University of Texas at Austin, Austin, TX, USA
e-mail: rauta@tacc.utexas.edu; lars@tacc.utexas.edu

M.M. Resch et al. (eds.), *Sustained Simulation Performance 2017*,
DOI 10.1007/978-3-319-66896-3_4

As a sample of evolution in the HPC landscape, consider the three flagship HPC systems provisioned by the Texas Advanced Computing Center (TACC) in the last 10 years: Ranger, Stampede, and Stampede2. Ranger debuted in the year 2008. It was equipped with AMD Opteron quad core processors, delivered approximately 579 TFLOPs of peak performance, and was in production for about 5 years. Stampede debuted in the year 2013 and is still in production. It can deliver approximately 10 PFLOPs of theoretical peak performance, and is equipped with Intel Sandy Bridge processors, Intel Knight's Corner coprocessors, and Nvidia K20 GPUs. The Stampede2 system, that is currently under development, will be equipped with the Intel Haswell processors, future generation Intel Xeon processors, and Intel Knight's Landing (KNL) processors. The estimated theoretical peak performance of Stampede2 is about 18 PFLOPs. Figure 1 depicts the rate of evolution in the HPC landscape considering the aforementioned systems deployed by TACC as examples.

For migrating applications from a system like Ranger to Stampede, and from Stampede to Stampede2, some level of software reengineering may be required to enhance the performance of the applications. The required reengineering may be in the form of increasing the level of parallelization in the applications by incorporating both OpenMP and MPI programming paradigms, or improving code vectorization to effectively take advantage of the Intel Sandy Bridge and Knight's Corner coprocessors, or optimizing the memory access pattern of the applications to benefit from the High-Bandwidth Memory (HBM) on the KNL processors.

As evident from the example of TACC resources, even though the current and future generation HPC systems may be equipped with high-end hardware components, they may not offer the same range of diversity in processing elements as compared to the previous generation systems. For example, unlike the Stampede system, the Stampede2 system does not include GPUs and Intel Knight's Corner

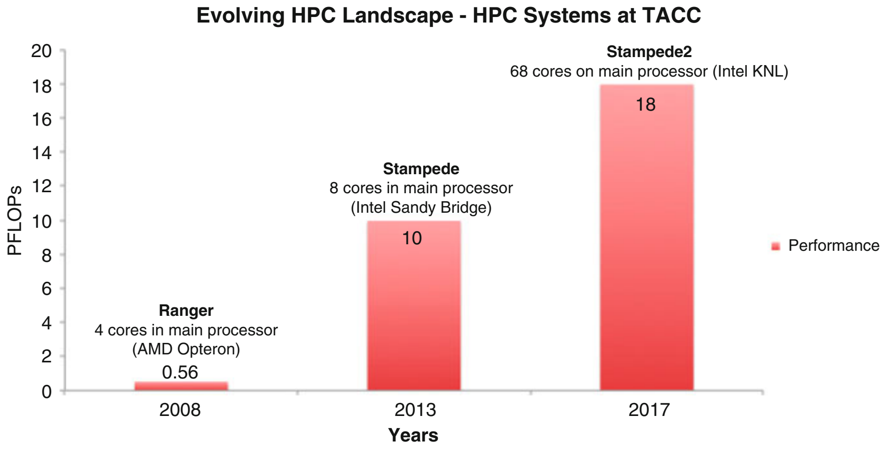


Fig. 1 Evolution in the HPC landscape

(KNC) coprocessors. Thus, the applications written using CUDA to take advantage of the GPUs, or offload programming model for KNC coprocessors will not be able to run on Stampede2. This implies that the HPC applications may need to be updated or reengineered at the same frequency as the systems on which they are supposed to run (typically, 4 years on an average).

Together with the evolution in the HPC hardware over the last several years, the parallel programming standards have also been continuously evolving by including new features for improved performance on modern HPC systems. For example, the nonblocking collective calls were added to the MPI 3.0 standard [1], and the `taskloop` construct was added to the OpenMP 4.5 standard [2]. Incorporation of such new features in the existing HPC applications requires time and effort in climbing the learning curve, and in reengineering the applications.

Given the discussion presented thus far, the questions that arise are:

- Are domain experts ready to invest time and effort in continuously modernizing their applications?
- Do we have enough trained workforce to support HPC code modernization and migration efforts?

During the process of pursuing the answers to the aforementioned questions, we found that the domain experts prefer to spend time in doing science rather than keeping up with the evolution in the HPC hardware and parallel programming standards. We also found that there is a shortfall of trained workforce in the area of parallel programming. Therefore, we need high-level tools for assisting users in their HPC code modernization and migration efforts. To address this need, we are developing high-level tools for supporting (1) the parallelization of serial applications using MPI, OpenMP, CUDA, and hybrid programming models, and (2) the migration of applications to the KNL processors. These tools are named as Interactive Parallelization Tool (IPT) and Interactive Code Adaptation Tool (ICAT). Before we present an overview of these tools, we explain the typical process of manual code modernization and migration, and share our perspective on the automation or semi-automation of the process. We also present a short overview of the KNL architecture and discuss some key considerations for porting applications to this architecture.

2 General Process for Manual Code Modernization and Migration

The traditional process of manually upgrading the code for taking advantage of the latest HPC systems is as follows:

1. Learn about a new hardware feature
2. Gauge the applicability of the hardware and the potential reward from the analysis of current bottlenecks (code profiling) and theoretical considerations

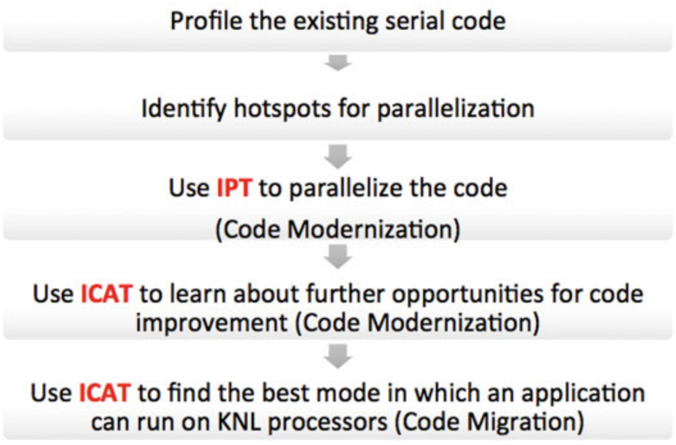

Profile the existing serial code
Identify hotspots for parallelization
Use IPT to parallelize the code (Code Modernization)
Use ICAT to learn about further opportunities for code improvement (Code Modernization)
Use ICAT to find the best mode in which an application can run on KNL processors (Code Migration)

Fig. 2 Using IPT and ICAT for code modernization

3. Develop a general concept of the necessary code modifications
4. Learn the syntax of the new programming model or the interface
5. Write a toy code
6. Insert pieces of the toy code into the target code for exploration and testing
7. Estimate performance gain (based on tests not on theoretical considerations) and judge the quality of the implementation
8. Modify, test, and release production code
9. Experiment with the various runtime options and environment variables
10. Learn good coding practices

Outlined above is a typical process and not all users go through all the steps in this process. The high-level tools that we are developing—IPT and ICAT—will assist the users in the aforementioned process, and in doing so, will significantly speed up many of its steps. IPT can assist the users in steps 1, 3–6, 8 and 10 mentioned above. ICAT can assist the users in steps 1 to 6, and 9–10. Figure 2 summarizes the steps for using IPT and ICAT together during the process of modernizing and migrating applications written in C, C++ and Fortran base languages.

3 Using IPT for Code Modernization (Parallelization)

IPT semi-automates the parallelization of existing C/C++ applications, and by doing so, helps in running the applications optimally on the latest HPC systems [1, 2]. It can support the parallelization of applications using any of the following parallel programming models: Message Passing Interface (MPI) [3], OpenMP [4], and CUDA [5]. IPT uses the specifications for parallelization as provided by the users (i.e., what to parallelize and where) and its knowledgebase of parallel programming

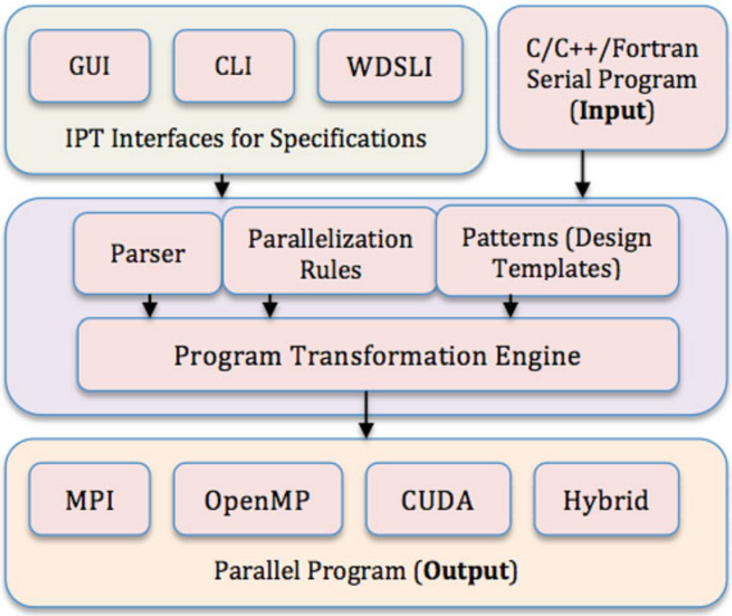

GUI
CLI
WDSLI
IPT Interfaces for Specifications
C/C++/Fortran Serial Program (Input)
Parser
Parallelization Rules
Patterns (Design Templates)
Program Transformation Engine
MPI
OpenMP
CUDA
Hybrid
Parallel Program (Output)

Fig. 3 Overview of IPT

expertise (encapsulated as design templates and rules), to generate parallel versions of the serial programs.

A high-level overview of the process of serial to parallel code transformation using IPT is shown in Fig. 3. As shown in Fig. 3, the user provides the serial code as input to IPT. This serial code is parsed by IPT, and as a next step, IPT prompts the user for additional input (or specifications). The user chooses the desired parallel programming model (MPI/OpenMP/CUDA) for their output code, applicable design patterns (e.g., stencil and pipeline), and hotspots for parallelization. The user also provides additional information when prompted by IPT for variable dependency analysis of the input serial application.

Using the infrastructure provided by a Program Transformation Engine (PTE) named ROSE [6], and user input, IPT analyzes and transforms the serial code into a parallel one. As required, it inserts, deletes and updates the program statements in the serial code for generating the parallel code. The generated parallel code is accessible to the user and is well annotated to give insights into the parallelization process. The design templates in IPT contain rules for parallelization and patterns for supporting data movement (e.g., data distribution, data collection, and data exchange) in MPI programs. During parallelization, IPT weaves these design templates into the serial code by the means of appropriate function calls.

We are currently working on (1) extending the capabilities of IPT for parallelizing additional categories of C/C++ applications (e.g., divide-and-conquer), (2) prototyping support for parallelizing Fortran applications, (3) adding support for parallelization using the hybrid programming model, and (4) making IPT accessible through a web-portal for convenient code generation and testing on computational resources of the national CyberInfrastructure (CI). In future, we will support a Graphical User Interface (GUI) and a Wizard-driven Domain-Specific Language Interface (WDSLI) to supplement the currently available Command-Line Interface (CLI) of IPT. The CLI and GUI are intended for parallelizing small applications interactively. However, working in these modes to provide parallelization requirements for large applications (over a couple of thousands of lines of code) can become tedious, and hence, WDSLI will be provided for capturing the parallelization requirements.

IPT can be used for self-paced learning of parallel programming, and in understanding the differences in the structure and performance of the parallel code generated for different specifications while using the same serial application.

4 Overview of KNL Processors

Before we discuss ICAT, we present a short overview of the Intel KNL processors. Intel KNL processors are equipped with 72 cores and have an extended memory architecture. The cores on these processors are organized in 36 pairs and each pair is known as a tile. These processors have a 16 GB High-Bandwidth Memory (HBM) called Multi-Channel DRAM (MCDRAM), alongside the traditional DDR4 memory that is approximately 400 GB [7].

4.1 Multiple Memory Modes

The MCDRAM can be configured for use in three different memory modes:

1. Cache mode: As a third-level cache that is under the control of the run-time system,
2. Flat mode: As an addressable memory like DDR4 that is under user control, or
3. Hybrid mode: Part of MCDRAM is configured in cache mode and part of it is configured in flat mode.

The cache mode is more convenient to use because it does not require any code modification or user interaction and ensures high performance for applications that have a small memory footprint. The applications having large memory footprints are likely to see a drop in their performance if they are run in cache mode due to frequent cache misses. It may be advantageous for such applications to manage the cache from within the code and to store only specific arrays in the MCDRAM—that is, using flat mode is recommended for such applications.

For selectively allocating arrays on MCDRAM, the existing code for dynamic memory allocation is modified to use special library calls or directives that are available through the HBWMALLOC interface [8]. In the case of C/C++ applications, the function calls for dynamic memory allocation—`calloc`, `malloc`, `realloc`, and `free` functions—are replaced with the analogous functions in the HBWMALLOC interface—`hbw_calloc`, `hbw_malloc`, `hbw_realloc`, and `hbw_free` functions. A header file for HBWMALLOC interface is also included. For allocating memory from MCDRAM in Fortran applications, a directive with the FASTMEM attribute is added after the declaration of the allocatable data structure of interest.

Understanding the concept of two memories (MCDRAM and DDR4), and doing the required code modifications for using MCDRAM effectively may not be difficult by itself. However, it takes time to understand the syntax of the code required for memory allocation on MCDRAM, to learn about the additional tools for understanding the application characteristics (cache miss or hit rate, sizes of memory objects etc.), and more importantly to derive the logic of the decision tree for allocating memory on MCDRAM or DDR4. In order to develop a portable code, it is also important to implement appropriate logic for handling situations that can give rise to runtime errors. For example, the code should handle situations where the user attempts to dynamically allocate more than 16 GB of memory on MCDRAM or tries to run the code on processors that do not support MCDRAM.

4.2 Multiple Cluster Modes

The tiles on a KNL processor are connected to each other with a mesh interconnect. Each core in a tile has its own L1 cache and a 1 MB L2 cache shared with the other core. The L2 cache on all the tiles are kept coherent with the help of a Distributed Tag Directory (DTD), organized as a set of per-tile Tag Directories (TDs). The TDs help in identifying the location and the state of cache lines on-die. When a memory request originates from a core, an appropriate TD handles it, and if needed passes the request to the right memory controller. All on-die communication for handling such memory requests happens over the mesh interconnect. To achieve low latency and high bandwidth of communication with caches, it is important that the on-die communication is kept as local as possible. For handling this on-die communication optimally, KNL processors can be configured in different cluster modes:

1. All-to-All: The memory addresses are uniformly distributed across all TDs, and this mode is used mainly for troubleshooting purposes or when other modes cannot be used because it can result in high latency for various on-die communication scenarios.
2. Quadrant or Hemisphere: The tiles on a processor are virtually divided into four parts called quadrants, and each quadrant is in proximity to a memory controller. The memory addresses controlled by the memory controller in a quadrant are mapped locally to the TD in that quadrant. This arrangement reduces the latency of a cache miss as compared to the all-to-all mode because the memory controller

and TD's are in the same locality. However, the TD servicing the memory request may not be local to the tile whose core initiated the memory request. The hemisphere mode is similar to the quadrant mode with the difference that the tiles on the chip are divided into two parts instead of four.

3. Sub-NUMA (SNC-4/SNC-2): Similar to the quadrant mode, the tiles are divided into four (SNC-4) or two (SNC-2) parts in this mode too. However, unlike in the quadrant mode, in the sub-NUMA mode, each part acts as a separate NUMA node. This means that, the core requesting memory access, the TD, and the memory channel for servicing the memory access request, are all in the same part.

While the quadrant mode could work well for majority of the applications, the sub-NUMA mode can result in better performance for multi-threaded NUMA-aware applications by pinning the threads and memory to the specific quadrants or hemispheres on each NUMA node. However, the users may have to do their own testing to find out the best cluster mode and the runtime options for their applications.

5 Using ICAT for Code Modernization and Migration (Porting Code to KNL Processors)

ICAT can assist users in porting their applications to KNL processors by helping them select the best memory mode and cluster mode and suggesting runtime options. It can reengineer their application code also to optimally take advantage of the MCDRAM while keeping it portable enough to run on other systems that do not support MCDRAM.

By using ICAT, a user can very quickly understand source code modifications, potential performance gains, and learn good coding practices for porting their applications to the KNL nodes. They can then move on to modifying their real application code using ICAT itself, or may cut-and-paste boilerplate code generated by ICAT. Thus, ICAT offers three key benefits to the users: (a) enables users to make a decision quickly regarding the best memory mode and cluster mode for their applications, (b) teaches good coding practices, and (c) assists in changing production code.

Figure 4 shows an overview of the functioning of ICAT, which is invoked from the command-line. ICAT prompts the user for input, such as, the name of an application's executable, path to the executable, and path to the application's source code. The user selects an appropriate advisor mode in which ICAT can run. The available advisor modes are: memory mode advisor, code adaptation advisor, cluster mode advisor, advanced vectorization advisor, and memory optimization advisor. ICAT performs memory usage and performance analyses by running the executable provided by the user with `perf` [9], and then if needed, with Vtune [10]. Metrics are also collected from the processes associated with the executable while it is

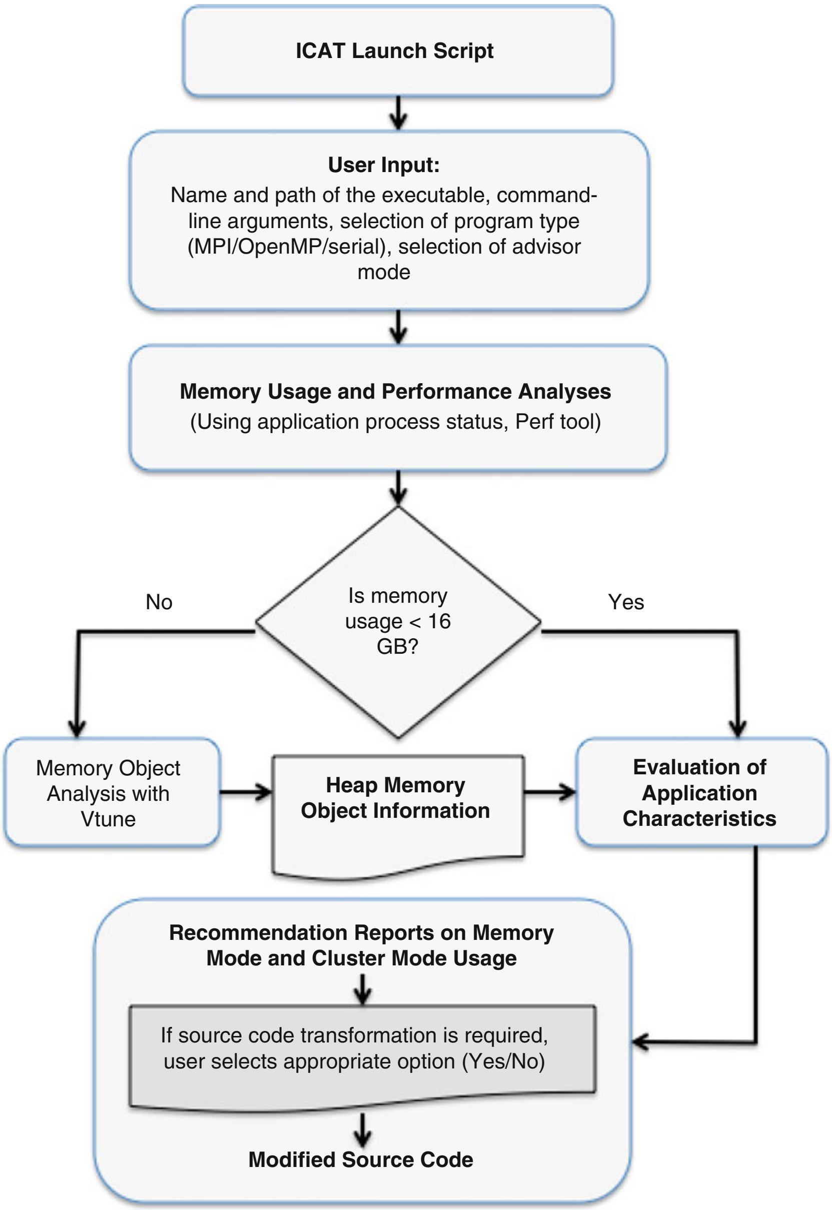

ICAT Launch Script
User Input:
Name and path of the executable, command-line arguments, selection of program type (MPI/OpenMP/serial), selection of advisor mode
Memory Usage and Performance Analyses
(Using application process status, Perf tool)
Is memory usage < 16 GB?
No
Yes
Memory Object Analysis with Vtune
Heap Memory Object Information
Evaluation of Application Characteristics
Recommendation Reports on Memory Mode and Cluster Mode Usage
If source code transformation is required, user selects appropriate option (Yes/No)
Modified Source Code

Fig. 4 Overview of functioning of ICAT

running. On the basis of the collected metrics and its analyses, ICAT generates recommendation reports for the user, and if needed, reengineers the application code.

ICAT can also be used for teaching and training activities related to the KNL processors. Following is how we envision using ICAT during a training session:

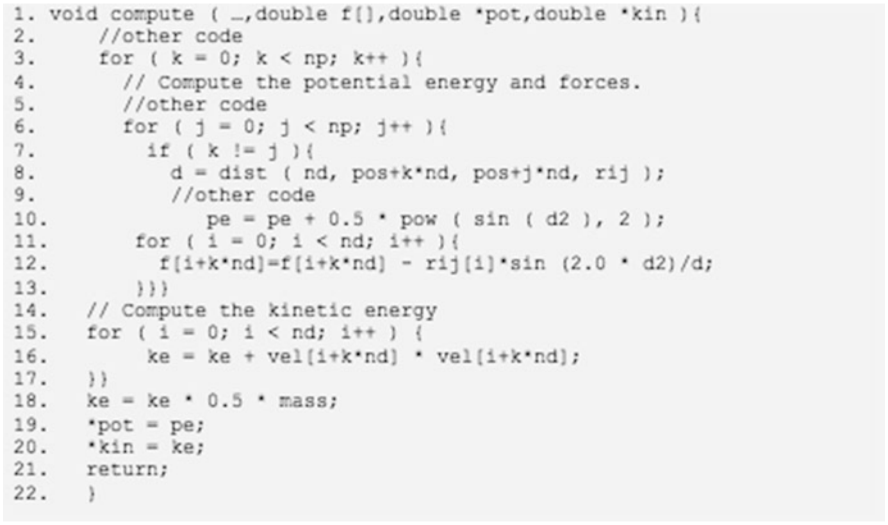
1. Explain the premise: HBM alongside the traditional memory; some raw performance comparisons (HBM v. DDR4); applicability for the 80–90% of cases that are neither I/O bound nor compute bound.
2. Start with a toy code from the sample code repository. Alternatively a small user code may be used in the future.
3. Run the toy code with the tool. The user will be guided through the modifications and will decide which arrays should be allocated on the HBM. These are the same decisions that the user will later make during the modification of the real-world applications.
4. Measure performance gain. Get a feel for the benefits of HBM, its limitations, and potential drawbacks.
5. Inspect the modified code and the syntax. Also understand how internally, i.e., in the code at runtime, decisions are being made and how a fallback is implemented for arrays that are too big for the HBM.

As part of the ongoing work, we are extending ICAT to support advanced vectorization and memory optimization. With these two features supported, ICAT will be able to help the users with tasks such as: reorganizing the data layout by changing array-of-structures to structures-of-arrays, converting scalars to vectors, and improving memory alignment of data structures.

6 Using IPT and ICAT with a Sample Application

To demonstrate the functionality of IPT and ICAT, let us consider a Molecular Dynamics (MD) simulation code. The code helps in following the path of particles that exert force on each other and are not constrained by any walls [11]. This MD code uses the velocity Verlet time integration scheme and the particles in the simulation interact with a central pair potential [11]. The compute-intensive steps in this test case are related to calculating force and energies in each time-step, as well as updating the values of the positions, velocities, and accelerations of the particles in the simulation.

A code snippet of the serial version of the MD simulation application is shown in Fig. 5 and the complete code can be accessed at [12]. In order to parallelize this code, the computations in the for-loop beginning at line # 3 of Fig. 5 should be distributed across multiple threads or processes. The values of the kinetic energy (`ke`) and potential energy (`pe`) are augmented in every iteration of this for-loop. Therefore, with the distribution of the iterations of the for-loop, only the partial values of `ke` and `pe` will be computed by each thread or process. Hence, all the partial values of



```
void compute ( …,double f[],double *pot,double *kin ){
    //other code
    for ( k = 0; k < np; k++ ){
      // Compute the potential energy and forces.
      //other code
      for ( j = 0; j < np; j++ ){
        if ( k != j ){
          d = dist ( nd, pos+k*nd, pos+j*nd, rij );
          //other code
             pe = pe + 0.5 * pow ( sin ( d2 ), 2 );
       for ( i = 0; i < nd; i++ ){
         f[i+k*nd]=f[i+k*nd] - rij[i]*sin (2.0 * d2)/d;
       }}}
   // Compute the kinetic energy
   for ( i = 0; i < nd; i++ ) {
        ke = ke + vel[i+k*nd] * vel[i+k*nd];
   }}
   ke = ke * 0.5 * mass;
   *pot = pe;
   *kin = ke;
   return;
   }
```

Fig. 5 Code snippet—MD simulation, serial version

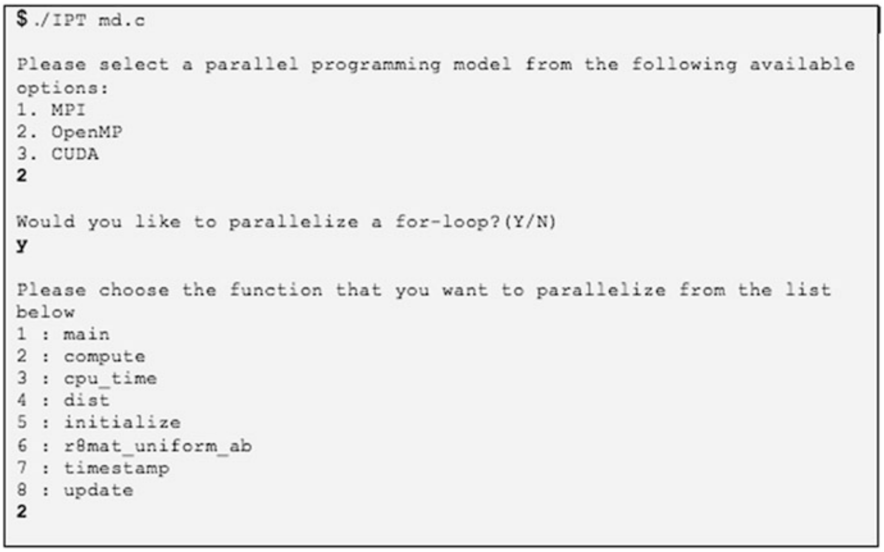
`ke` and `pe` computed using the multiple independent threads and processes should be combined meaningfully to obtain accurate results. For combining the partial values of `ke` and `pe`, a reduction operation is needed.

6.1 Using IPT to Parallelize the MD Application

In this section, we will demonstrate the usage of IPT by generating an OpenMP version of the serial MD simulation application. As shown in Fig. 6, IPT is invoked from the command-line, and the path to the file containing the serial code is provided. Next, a parallel programming model is selected, here OpenMP. This is followed by selecting the function that contains the hotspot for parallelization from the list of functions presented by IPT.

IPT analyzes the code in the function selected by the user (see Fig. 6), and presents a list of the for-loops that can be parallelized (because, in this example, the user chose to parallelize for-loops). As shown in Fig. 7, users can accept or decline to parallelize the for-loops presented by IPT for parallelization.

For constructing the clauses of the OpenMP directives, IPT can detect the variables that should be part of the `shared`, `private`, and `firstprivate` clauses. However, IPT relies on the user-guidance for constructing the reduction clause of the relevant OpenMP directives (`#pragma omp parallel`, or `#pragma omp parallel for`, or `#pragma omp for`). Reduction variables are the variables



```
$ ./IPT md.c

Please select a parallel programming model from the following available
options:
1. MPI
2. OpenMP
3. CUDA
2

Would you like to parallelize a for-loop?(Y/N)
y

Please choose the function that you want to parallelize from the list
below
1 : main
2 : compute
3 : cpu_time
4 : dist
5 : initialize
6 : r8mat_uniform_ab
7 : timestamp
8 : update
2
```

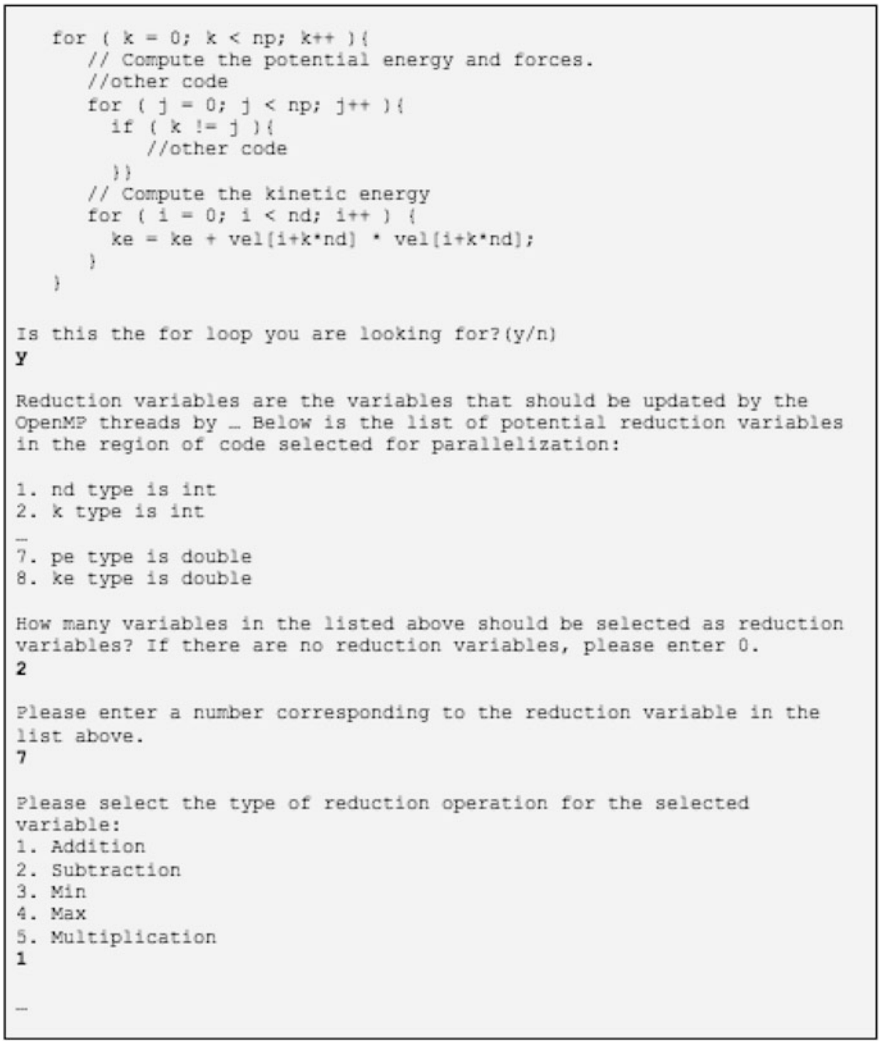
Fig. 6 Invoking IPT for parallelizing MD simulation application

that should be updated by the OpenMP threads by creating a private copy for each reduction variable and initializing them for each thread. The values of the variables from each thread are combined according to a mathematical operation like sum, multiplication, etc. and the final result is written to a global shared variable. IPT generates a list of potential variables that can be part of a reduction clause, and prompts the user to select the relevant variables and appropriate reduction operation.

In some cases, IPT is unable to analyze the pattern related to updating the array elements at the hotspot for parallelization. This typically happens when multiple levels of indirection are involved during the process of updating the values of the array elements. In such situations, as shown in Fig. 8, IPT relies on the user to provide additional information on the nature of the update operation on their array elements.

IPT also prompts the user to confirm if the I/O in their application should be happening from a single thread (or process) or using all the threads (or processes involved in the computation). If there is any region of code that should not be executed in parallel, then, the user can inform IPT about this as well.

A snippet of the parallelized version of the MD simulation application is shown in Fig. 9. In addition to updating the code at the hotspot for parallelization by inserting OpenMP directives (lines 3–5 of Fig. 9), IPT inserts appropriate library header files as well.

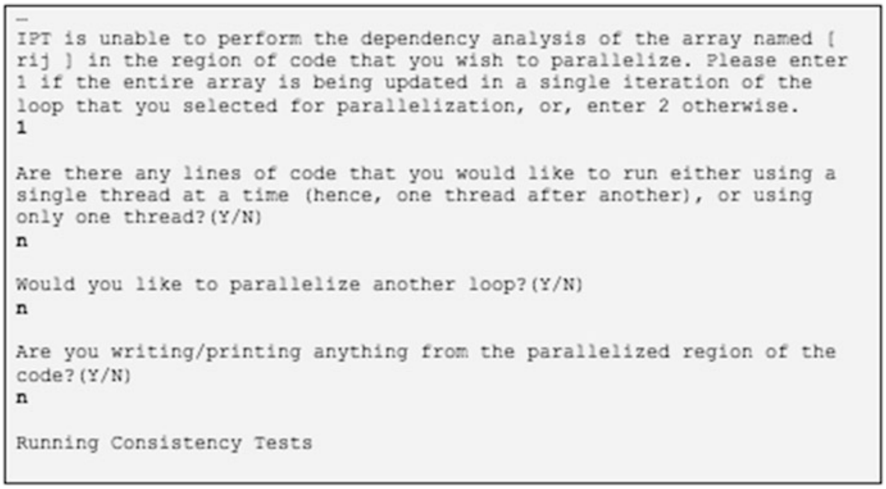


```
    for ( k = 0; k < np; k++ ){
       // Compute the potential energy and forces.
       //other code
       for ( j = 0; j < np; j++ ){
         if ( k != j ){
            //other code
         }}
       // Compute the kinetic energy
       for ( i = 0; i < nd; i++ ) {
         ke = ke + vel[i+k*nd] * vel[i+k*nd];
       }
    }

Is this the for loop you are looking for?(y/n)
y

Reduction variables are the variables that should be updated by the
OpenMP threads by … Below is the list of potential reduction variables
in the region of code selected for parallelization:

1. nd type is int
2. k type is int
…
7. pe type is double
8. ke type is double

How many variables in the listed above should be selected as reduction
variables? If there are no reduction variables, please enter 0.
2

Please enter a number corresponding to the reduction variable in the
list above.
7

Please select the type of reduction operation for the selected
variable:
1. Addition
2. Subtraction
3. Min
4. Max
5. Multiplication
1

…
```

Fig. 7 User guiding IPT in selecting the reduction variables

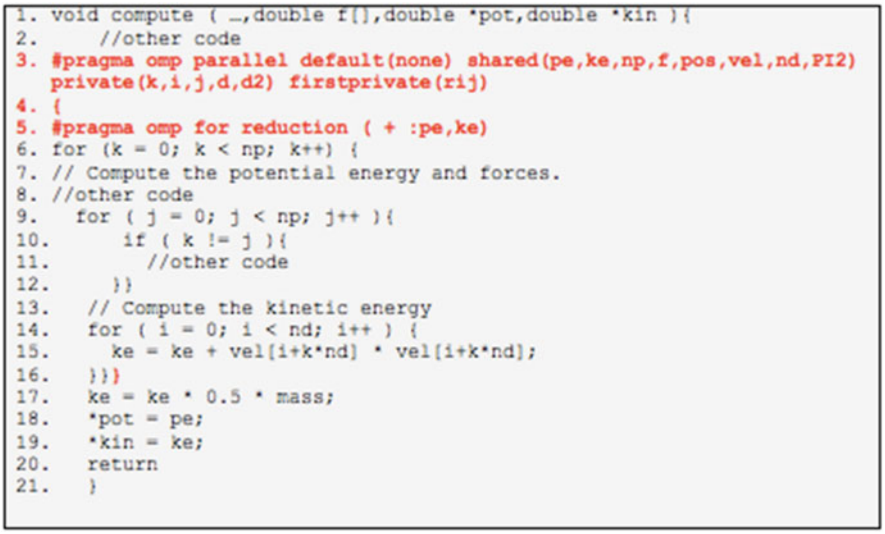
6.2 *Using ICAT to Adapt the MD Application for KNL Processors*

As described in Sect. 3, before running an application on KNL processors, it is important to understand the application's characteristics so that the best memory mode and the cluster mode configuration of the KNL processors can be selected for it. Depending upon the memory needs of the application, some reengineering may also be required for selectively allocating memory for specific arrays on MCDRAM.



```
_
IPT is unable to perform the dependency analysis of the array named [
rij ] in the region of code that you wish to parallelize. Please enter
1 if the entire array is being updated in a single iteration of the
loop that you selected for parallelization, or, enter 2 otherwise.
1

Are there any lines of code that you would like to run either using a
single thread at a time (hence, one thread after another), or using
only one thread?(Y/N)
n

Would you like to parallelize another loop?(Y/N)
n

Are you writing/printing anything from the parallelized region of the
code?(Y/N)
n

Running Consistency Tests
```

Fig. 8 User guiding IPT in analyzing the nature of the updates made to the array values

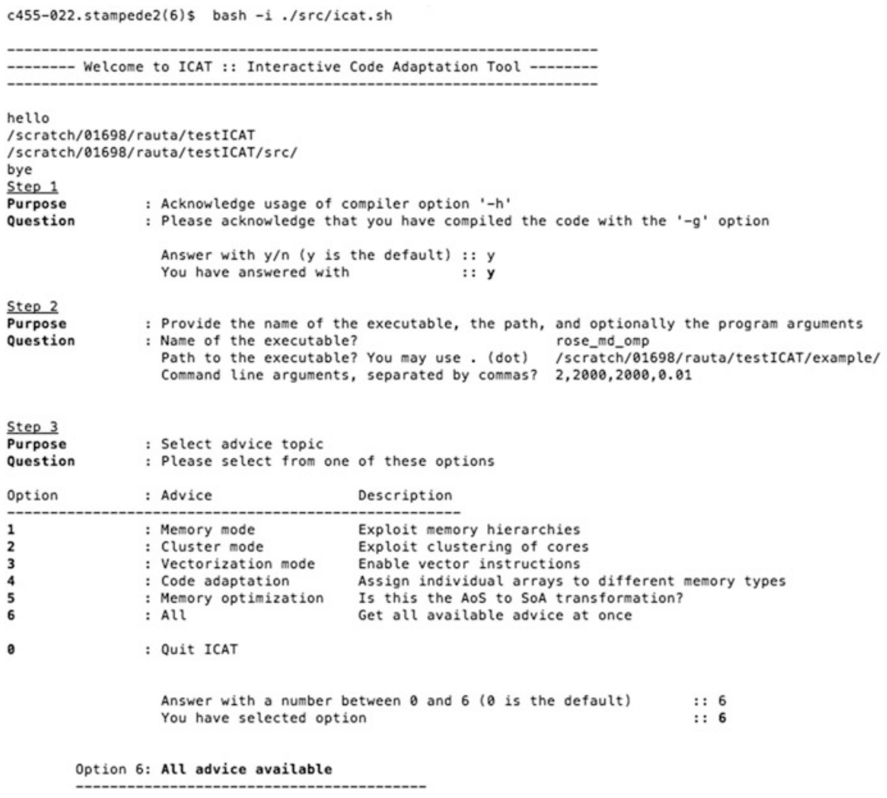


```
void compute ( …,double f[],double *pot,double *kin ){
    //other code
#pragma omp parallel default(none) shared(pe,ke,np,f,pos,vel,nd,PI2)
private(k,i,j,d,d2) firstprivate(rij)
{
#pragma omp for reduction ( + :pe,ke)
for (k = 0; k < np; k++) {
// Compute the potential energy and forces.
//other code
  for ( j = 0; j < np; j++ ){
      if ( k != j ){
        //other code
     }}
   // Compute the kinetic energy
   for ( i = 0; i < nd; i++ ) {
     ke = ke + vel[i+k*nd] * vel[i+k*nd];
   }}}
   ke = ke * 0.5 * mass;
   *pot = pe;
   *kin = ke;
   return
   }
```

Fig. 9 Snippet of OpenMP code generated by IPT—MD Simulation application

We demonstrate the usage of ICAT as a decision-support system by using it for porting the OpenMP version of the MD simulation application to KNL processors.

Before invoking ICAT, we compiled the OpenMP version of the MD simulation application with the -g flag. After invoking ICAT from the command-line, as shown in Fig. 10, we provide the path to the application executable and the arguments required to run it. We also select the advisor mode in which ICAT should run. Using this information, ICAT first profiles the application by running it in real-time



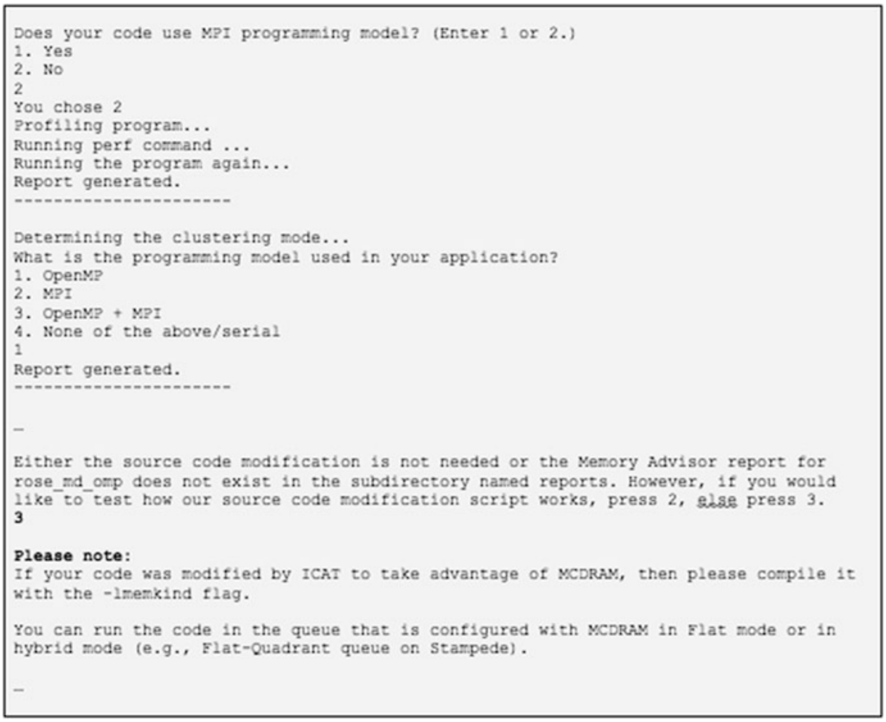
```
c455-022.stampede2(6)$  bash -i ./src/icat.sh

--------------------------------------------------------------------
-------- Welcome to ICAT :: Interactive Code Adaptation Tool --------
--------------------------------------------------------------------

hello
/scratch/01698/rauta/testICAT
/scratch/01698/rauta/testICAT/src/
bye
Step 1
Purpose         : Acknowledge usage of compiler option '-h'
Question        : Please acknowledge that you have compiled the code with the '-g' option

                  Answer with y/n (y is the default) :: y
                  You have answered with             :: y

Step 2
Purpose         : Provide the name of the executable, the path, and optionally the program arguments
Question        : Name of the executable?                   rose_md_omp
                  Path to the executable? You may use . (dot)   /scratch/01698/rauta/testICAT/example/
                  Command line arguments, separated by commas?  2,2000,2000,0.01


Step 3
Purpose         : Select advice topic
Question        : Please select from one of these options

Option          : Advice               Description
-----------------------------------------------------------
1               : Memory mode          Exploit memory hierarchies
2               : Cluster mode         Exploit clustering of cores
3               : Vectorization mode   Enable vector instructions
4               : Code adaptation      Assign individual arrays to different memory types
5               : Memory optimization  Is this the AoS to SoA transformation?
6               : All                  Get all available advice at once

0               : Quit ICAT


                  Answer with a number between 0 and 6 (0 is the default)      :: 6
                  You have selected option                                      :: 6


        Option 6: All advice available
        ----------------------------------------
```

Fig. 10 Invoking ICAT from the command-line

and gathers the application's characteristics. It then generates a recommendation report regarding the appropriate memory mode for the application and instructions for compiling the code.

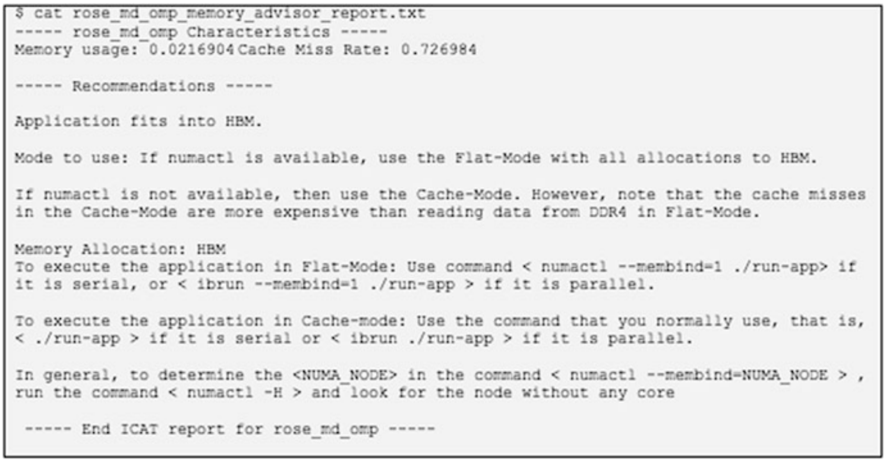
Using the memory mode report that it generated and the input regarding the programming model of the application, ICAT also generates a report with the recommendation for the cluster mode to use. Then, as shown in Fig. 11, ICAT informs that the entire MD simulation application fits in the MCDRAM. Hence, no code adaptation is required. However, if the user still wants to see how the code would be adapted to selectively use the MCDRAM, they may choose to do so by selecting the appropriate option while ICAT is running.

The reports generated by the memory mode advisor and the cluster mode advisor are shown in Figs. 12 and 13. For the OpenMP version of the MD simulation application, ICAT recommends running the application on the KNL node which has the MCDRAM configured in flat-mode if `numactl` is available. If `numactl` is not available, it recommends running the application on the KNL node which has the MCDRAM configured in cache-mode. For the cluster mode, ICAT recommends using the SNC-4 configuration. In the case of Stampede2 system,



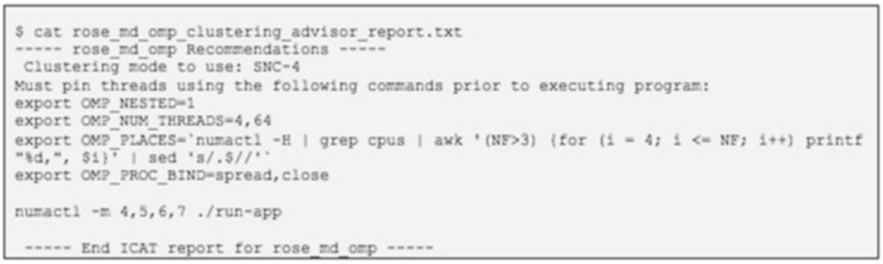
```
Does your code use MPI programming model? (Enter 1 or 2.)
1. Yes
2. No
2
You chose 2
Profiling program...
Running perf command ...
Running the program again...
Report generated.
---------------------

Determining the clustering mode...
What is the programming model used in your application?
1. OpenMP
2. MPI
3. OpenMP + MPI
4. None of the above/serial
1
Report generated.
---------------------


_

Either the source code modification is not needed or the Memory Advisor report for
rose_md_omp does not exist in the subdirectory named reports. However, if you would
like to test how our source code modification script works, press 2, else press 3.
3

Please note:
If your code was modified by ICAT to take advantage of MCDRAM, then please compile it
with the -lmemkind flag.

You can run the code in the queue that is configured with MCDRAM in Flat mode or in
hybrid mode (e.g., Flat-Quadrant queue on Stampede).


_
```

Fig. 11 ICAT running in different advisor modes



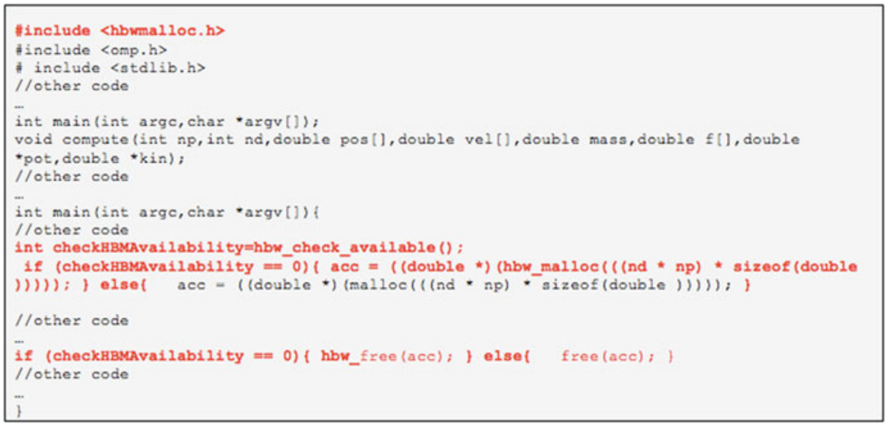
```
$ cat rose_md_omp_memory_advisor_report.txt
----- rose_md_omp Characteristics -----
Memory usage: 0.0216904 Cache Miss Rate: 0.726984

----- Recommendations -----

Application fits into HBM.

Mode to use: If numactl is available, use the Flat-Mode with all allocations to HBM.

If numactl is not available, then use the Cache-Mode. However, note that the cache misses
in the Cache-Mode are more expensive than reading data from DDR4 in Flat-Mode.

Memory Allocation: HBM
To execute the application in Flat-Mode: Use command < numactl --membind=1 ./run-app> if
it is serial, or < ibrun --membind=1 ./run-app > if it is parallel.

To execute the application in Cache-mode: Use the command that you normally use, that is,
< ./run-app > if it is serial or < ibrun ./run-app > if it is parallel.

In general, to determine the <NUMA_NODE> in the command < numactl --membind=NUMA_NODE > ,
run the command < numactl -H > and look for the node without any core

 ----- End ICAT report for rose_md_omp -----
```

Fig. 12 Memory mode advisor report



```
$ cat rose_md_omp_clustering_advisor_report.txt
----- rose_md_omp Recommendations -----
 Clustering mode to use: SNC-4
Must pin threads using the following commands prior to executing program:
export OMP_NESTED=1
export OMP_NUM_THREADS=4,64
export OMP_PLACES=`numactl -H | grep cpus | awk '(NF>3) {for (i = 4; i <= NF; i++) printf
"%d,", $i}' | sed 's/.$//'`
export OMP_PROC_BIND=spread,close

numactl -m 4,5,6,7 ./run-app

 ----- End ICAT report for rose_md_omp -----
```

Fig. 13 Cluster mode advisor report



```
#include <hbwmalloc.h>
#include <omp.h>
# include <stdlib.h>
//other code
...
int main(int argc,char *argv[]);
void compute(int np,int nd,double pos[],double vel[],double mass,double f[],double
*pot,double *kin);
//other code
...
int main(int argc,char *argv[]){
//other code
int checkHBMAvailability=hbw_check_available();
 if (checkHBMAvailability == 0){ acc = ((double *)(hbw_malloc(((nd * np) * sizeof(double
))))); } else{   acc = ((double *)(malloc(((nd * np) * sizeof(double ))))); }

//other code
...
if (checkHBMAvailability == 0){ hbw_free(acc); } else{   free(acc); }
//other code
...
}
```

Fig. 14 Code modifications done using ICAT

the aforementioned recommendations imply that the OpenMP version of the MD simulation application should be run on the KNL node in the "Flat-SNC4" queue.

As can be noticed from Fig. 11, ICAT recommended against modifying the OpenMP version of the MD simulation application to use the HBWMALLOC interface. However, if a user still wishes to modify the application code to use the HBWMALLOC interface, they may do so using ICAT. A snippet of the modified version of the MD simulation application produced using ICAT is shown in Fig. 14. The modifications made by ICAT include: inserting code for including the `hbwmalloc.h` file, checking the availability of MCDRAM in the underlying architecture, replacing the call/s to the `malloc` function with the `hbw_malloc` function, and replacing the call/s to the `free` function with the `hbw_free` function call/s.

7 Conclusion

HPC system hardware and the programming models are constantly evolving. Sometimes the changes are big, but often the changes are incremental. Even if most users are not aiming for peak performance, they need to spend some effort in modernizing their applications to keep up with the major developments. Not modernizing their code base to keep up with the technology is not viable because with inefficient code, the researchers will neither be competitive (a) in the scientific arena to handle larger problem sizes and calculations than what they are doing now, nor (b) when HPC resources are allocated at open-science data centers [13]. Often, users attempt to strike a balance between effort and reward and they cannot afford to explore all possible avenues for code modernization at a given point in time. Therefore, high-level tools — like IPT and ICAT — that are geared towards assisting the users in code modernization and migration efforts on the latest HPC platforms are needed.

Acknowledgements We are very grateful to the National Science Foundation for grant # 1642396, ICERT REU program (National Science Foundation grant # 1359304), XSEDE (National Science Foundation grant # ACI-1053575), and TACC for providing resources required for this project. We are grateful to our students (Madhav Gupta, Trung Nguyen Ba, Alex Suryapranata, Julio Olaya, Tiffany Connors, Ejenio Capetillo, and Shweta Gulati) for their contributions to the IPT, ITALC, and ICAT codebase. We are also grateful to Dr. Purushotham Bangalore for providing guidance and code templates for developing FraSPA, which was the precursor of IPT.

References

1. Arora, R., Olaya, J., Gupta, M.: A tool for interactive parallelization. In: Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment (XSEDE'14), Article 51, p. 8. ACM, New York (2014). http://dx.doi.org/10.1145/2616498.2616558
2. Arora, R., Koesterke, L.: Interactive code adaptation tool for modernizing applications for Intel Knights Landing processors. In: Proceedings of the 2017 Conference on the Practice & Experience in Advanced Research Computing (PEARC17). ACM, New York (2017) http://dx.doi.org/10.1145/3093338.3093352
3. MPI: A Message Passing Interface Standard. Message Passing Interface Forum (2015). http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. Cited 16 June 2017
4. OpenMP Application Programming Interface (2015). http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf. Cited 16 June 2017
5. CUDA Toolkit Documentation (2017). http://docs.nvidia.com/cuda/#axzz4lGuUKK2x. Cited 16 June 2017
6. ROSE User Manual (2017). http://rosecompiler.org/ROSE_UserManual/ROSE-UserManual.pdf. Cited 16 June 2017
7. Sodani, A.: Knights landing (KNL): 2nd generation Intel® Xeon Phi processor. In: 2015 IEEE Hot Chips 27 Symposium (HCS) (2015). doi:10.1109/HOTCHIPS.2015.7477467
8. Intel Corporation HBWMALLOC (2015). https://www.mankier.com/3/hbwmalloc. Cited 16 June 2017

9. perf: Linux Profiling with Performance Counters. https://perf.wiki.kernel.org/index.php/Main_Page. Cited 16 June 2017
10. Vtune Performance Profiler (2017). https://software.intel.com/en-us/intel-vtune-amplifier-xe. Cited 16 June 2017
11. Rapaport, D.: An introduction to interactive molecular-dynamics simulation. Comput. Phys. **11**(4), 337–347 (1997)
12. Molecular Dynamics Code. http://people.sc.fsu.edu/~jburkardt/c_src/md/md.c. Cited 16 June 2017
13. XSEDE Allocations Overview. https://www.xsede.org/allocations. Cited 16 June 2017