Chapter 1

Exploring Use of the Reserved Core

John Holmen[1], Alan Humphrey[2], Martin Berzins[3]

[1]SCI Institute & School of Computing, University of Utah

In this chapter, we illustrate benefits of thinking in terms of thread management techniques when using a centralized scheduler model along with interoperability of MPI and PThreads. This is facilitated through an exploration of thread placement strategies for an algorithm modeling radiative heat transfer with special attention to the 61^{st} core. This algorithm plays a key role within the Uintah Computational Framework (UCF) and current efforts taking place at the University of Utah to model next-generation, large-scale clean coal boilers. In such simulations, this algorithm models the dominant form of heat transfer and consumes a large portion of compute time. Exemplified by a real-world example, this chapter presents our early efforts in porting a key portion of a scalability-centric codebase to the Intel® Xeon Phi™ coprocessor. Specifically, this chapter presents results from our experiments profiling the native execution of a reverse Monte-Carlo ray tracing-based radiation model on a single coprocessor. These results demonstrate that our fastest run configurations utilized the 61^{st} core and that performance was not profoundly impacted when explicitly oversubscribing the coprocessor operating system thread. Additionally, this chapter presents a portion of radiation model source code, a MIC-centric UCF cross-compilation example, and less conventional thread management techniques for developers utilizing the PThreads threading model.

The Uintah Computational Framework

Regularly maintained as open-source software (MIT License), the Uintah Computational Framework (UCF) consists of a set of simulation components and libraries facilitating the simulation and analysis of complex chemical and physical

reactions. These reactions are modeled by solving partial differential equations on structured adaptive mesh refinement grids. This framework was originally developed as part of the University of Utah's Center for the Simulation of Accidental Fires and Explosions (C-SAFE) initiative in 1997. Since then, it has been widely ported and used to develop novel techniques for understanding large pool eddy fires and new methods for simulating fluid-structure interactions.

Released in January of 2015, UCF release 1.6.0 features four primary simulation components:

- *ARCHES*: This component targets the simulation of turbulent reacting flows with participating media radiation.
- *ICE*: This component targets the simulation of both low-speed and high-speed compressible flows.
- *MPM*: This component targets the simulation of multi-material, particle-based structural mechanics.
- *MPM-ICE*: This component corresponds to the combination of the ICE and MPM components for the simulation of fluid-structure interactions.

Though small-scale simulations are supported, the UCF emphasizes large-scale simulations. This in mind, the UCF must be able to leverage the increasing adoption of the Intel MIC Architecture within current and emerging heterogeneous supercomputer architectures. Aligning with this need, our research efforts have targeted porting the UCF's reverse Monte-Carlo ray tracing-based radiation model to this architecture. This radiation model has been chosen to support the University of Utah's Carbon Capture Multi-Disciplinary Simulation Center's (CCMSC) goal of simulating a 350 MWe clean coal boiler being developed by Alstom Power.

Radiation Modeling with the UCF

ARCHES was initially developed using the parallel discrete ordinates method and P1 approximation to aid in solving the radiative transport equation. Though scalable, this approach resulted in solution of the associated linear systems being the main computational cost of reacting flow simulations.

To reduce this cost, recent attention has been given to exploring more efficient reverse Monte-Carlo ray tracing (RMCRT) methods. These efforts have resulted in the development of a stand-alone RMCRT model capable of being used with any of the UCF's simulation components.

RMCRT leverages reciprocity in radiative transfer and Monte-Carlo methods. The resulting approach samples a simulation mesh by tracing rays *towards* their origin. During traversal, the amount of incoming radiative intensity absorbed by the origin is computed to aid in solving the radiative transport equation.

This approach creates the potential for scalable parallelism as multiple rays can be traced simultaneously at any given cell and/or timestep. Additionally, it

also eliminates the need to trace rays that may never reach a given origin. Note however, this is achieved at the expense of an all-to-all communication between timesteps.

For these experiments, the primary hotspot was a function known as **updateSumI()**. Critical to RMCRT, this function traverses rays using a ray marching algorithm while simultaneously computing each ray's contribution to the incoming radiative intensity. Putting this into perspective, upwards of 200 million rays were cast at each compute timestep during these experiments.

Figure 1.1 and Figure 1.2 present portions of code corresponding to **updateSumI()**. Referring to the version of Uintah used for this chapter, **updateSumI()** can be found in **RMCRTCommon.cc**. Note that, Figure 1.1 and Figure 1.2 have been abbreviated so as not to include special scenarios that were not explored during these experiments.

To learn more about radiative heat transfer and the UCF's radiation models, please refer to the *For More Information* section.

Cross-Compiling the UCF

For this port, cross-compilation and resolution of third party dependencies have been the primary challenges.

When preparing to build the UCF, a developer must first cross-compile any required libraries that are unavailable on the coprocessor or feature conflicting versions between architectures. For this work, the UCF required cross-compilation of *libxml2* and *zlib* as demonstrated in Figure 1.3 and Figure 1.4.

After resolving dependencies, a developer then constructs a configure line denoting how to build the UCF executable. Figure 1.5 demonstrates how to cross-compile the UCF with utilities only.

With cross-compilation complete, a developer must then transfer native MIC executables, UCF problem specifications, and dynamically linked libraries to the coprocessor.

To learn more about obtaining, installing, and using the UCF, please refer to the *For More Information* section.

Towards Demystifying the Reserved Core

On the 61-core coprocessor, the last-most physical core contains logical cores 241, 242, 243, and 0. Though */proc/cpuinfo* core id: 60 in practice, this physical core is commonly referred to as the 61^{st} core. The 61^{st} core is unique in that logical core 0 is reserved for the Xeon Phi operating system. Additionally, the 61^{st} core is also reserved for the offload daemon. While it is reportedly safe to use all 244 threads for native execution, this begs the question - *How does one effectively manage the* 61^{st} *core when executing natively?*

To explore this question, we have experimented with a number of thread placement strategies featuring varying affinity patterns and thread counts. De-

```
RMCRTCommon::updateSumI ( Vector&               ray_direction,
                          Vector&               ray_location,
                          const IntVector&      origin,
                          const Vector&         Dx,
                          constCCVariable< T >& sigmaT4OverPi,
                          constCCVariable< T >& abskg,
                          constCCVariable<int>& celltype,
                          unsigned long int&    nRaySteps,
                          double&               sumI,
                          MTRand&               mTwister) {

  IntVector cur      = origin;
  IntVector prevCell = cur;

  // Step and sign used for ray marching
  int  step[3]; // Gives +1 or -1 based on sign
  bool sign[3];

  // Update step and sign to determine whether dimensions are incremented
  // or decremented as cell boundaries are crossed
  Vector inv_ray_direction = Vector( 1.0 ) / ray_direction;
  findStepSize( step, sign, inv_ray_direction );
  Vector D_DxRatio( 1, Dx.y() / Dx.x(), Dx.z() / Dx.x() );

  // Compute the distance from the ray origin to the nearest cell boundary
  // for a given dimension
  Vector tMax;
  tMax.x( (origin[0] + sign[0] - ray_location[0]) *
          inv_ray_direction[0] );
  tMax.y( (origin[1] + sign[1] * D_DxRatio[1] - ray_location[1]) *
          inv_ray_direction[1] );
  tMax.z( (origin[2] + sign[2] * D_DxRatio[2] - ray_location[2]) *
          inv_ray_direction[2] );

  // Compute the distance required to traverse a single cell
  Vector tDelta = Abs( inv_ray_direction ) * D_DxRatio;

  // Intialize per-ray variables
  bool   in_domain            = true;
  double tMax_prev            = 0;
  double intensity            = 1.0;
  double fs                   = 1.0;
  double optical_thickness    = 0;
  double expOpticalThick_prev = 1.0;

  // Traverse a given ray until the incoming intensity that would arrive
  // at the origin falls below a user-defined threshold
  while ( intensity > d_threshold ) {

    DIR face = NONE;

    // Traverse a given ray until it leaves the simulation mesh
    while ( in_domain ) {

      prevCell = cur;
      double disMin = -9; // Represents ray segment length

      T abskg_prev = abskg[prevCell];
      T sigmaT4OverPi_prev = sigmaT4OverPi[prevCell];
```

Figure 1.1: RMCRT-based radiation modeling hotspot

```
      // Determine which cell the ray will enter next
      if ( tMax[0] < tMax[1] ) {   // X < Y
        if ( tMax[0] < tMax[2] ) { // X < Z
          face = X;
        } else {
          face = Z;
        }
      } else {
        if ( tMax[1] < tMax[2] ) { // Y < Z
          face = Y;
        } else {
          face = Z;
        }
      }

      // Update ray marching variables
      cur[face]  = cur[face] + step[face];
      disMin     = ( tMax[face] - tMax_prev );
      tMax_prev  = tMax[face];
      tMax[face] = tMax[face] + tDelta[face];

      // Update ray location
      ray_location[0] = ray_location[0] + ( disMin * ray_direction[0] );
      ray_location[1] = ray_location[1] + ( disMin * ray_direction[1] );
      ray_location[2] = ray_location[2] + ( disMin * ray_direction[2] );

      // Check if the cell is in the simulation mesh
      in_domain = ( celltype[cur] == d_flowCell );

      optical_thickness += Dx.x() * abskg_prev * disMin;

      nRaySteps++;

      double expOpticalThick = exp(-optical_thickness );

      sumI += sigmaT4OverPi_prev *
              ( expOpticalThick_prev - expOpticalThick ) *
              fs;

      expOpticalThick_prev = expOpticalThick;

    } // End of in_domain while loop

    T wallEmissivity = abskg[cur];

    // Ensure that wall emissivity does not exceed one
    if ( wallEmissivity > 1.0 ) {
      wallEmissivity = 1.0;
    }

    intensity = exp( -optical_thickness );

    sumI += wallEmissivity * sigmaT4OverPi[cur] * intensity;

    intensity = intensity * fs;

    // Terminate a ray upon reaching mesh boundaries
    if(!d_allowReflect) intensity = 0;

  }  // End of intensity while loop
} // End of updateSumI function
```

Figure 1.2: RMCRT-based radiation modeling hotspot (continued)

```
wget http://xmlsoft.org/sources/libxml2-2.7.8.tar.gz
tar xvf libxml2-2.7.8.tar.gz
cd libxml2-2.7.8
./configure \
  --prefix=$HOME/installs/mic/libxml2-2.7.8 \
  --host=x86_64-k1om-linux \
  --enable-static \
  --without-python \
  CC=icc \
  CXX=icpc \
  CFLAGS="-mmic" \
  CXXFLAGS="-mmic" \
  LDFLAGS="-mmic"
make -j32 all
make install
```

Figure 1.3: Cross-compiling *libxml2-2.7.8* for the coprocessor

```
wget http://zlib.net/zlib-1.2.8.tar.gz
tar xvf zlib-1.2.8.tar.gz
cd zlib-1.2.8
CC=icc CXX=icpc CFLAGS="-mmic" CXXFLAGS="-mmic" LDFLAGS="-mmic" \
  ./configure \
  --prefix=$HOME/installs/mic/zlib-1.2.8 \
  --static
make -j32 all
make install
```

Figure 1.4: Cross-compiling *zlib-1.2.8* for the coprocessor

```
../src/configure \
  --host=x86_64-k1om-linux \
  --enable-64bit \
  --enable-optimize="-O2 -mkl=parallel -mmic -mt_mpi" \
  --enable-assertion-level=0 \
  --enable-static \
  --with-libxml2=$HOME/installs/mic/libxml2-2.7.8 \
  --with-mpi=/opt/intel/impi/5.0.1.035/mic \
  --with-zlib=$HOME/installs/mic/zlib-1.2.8 \
  CC=mpiicc \
  CXX=mpiicpc \
  F77=mpiifort
```

Figure 1.5: Configure line for a MIC-specific UCF build

tailed information regarding the parameters explored can be found in the subsequent subsections and the *Simulation Configuration* section. These parameters have been chosen to examine a multitude of strategies for handling the 61^{st} core.

Exploring Thread Affinity Patterns

Structured around a task-based, MPI+PThreads parallelism model, the UCF features a task scheduler component. This component is responsible for computing task dependencies, determining task execution order, and ensuring correctness of inter-process communication.

As of UCF release 1.6.0, the default multi-threaded scheduler is the Threaded MPI Scheduler. This dynamic scheduler features non-deterministic, out-of-order task execution at runtime. This is facilitated using a control thread and *nThreads*-1 task execution threads, where *nThreads* equals the number of threads launched at runtime.

Given this centralized model, we have experimented with the affinity patterns described below. Referring to the version of Uintah used for this chapter, implementations of these affinity patterns can be found in **ThreadedMPIScheduler.cc**. Note that, nt corresponds to the number of threads per physical core.

- *Compact*: This pattern binds task execution threads incrementally across logical cores 1 through nt in a given physical core first and then across physical cores 1 through 61. This pattern is modeled after OpenMP's **KMP_AFFINITY = compact** with values of **61c,2t**, **61c,3t**, and **61c,4t** for the **KMP_PLACE_THREADS** environment variable.

- *None*: This pattern allows both the control and task execution threads to run anywhere among all 244 logical cores.

- *Scatter*: This pattern binds task execution threads incrementally across physical cores 1 through 60 first and then across logical cores 1 through nt in a given physical core. This pattern is modeled after OpenMP's **KMP_AFFINITY = scatter** with values of **60c,2t**, **60c,3t**, and **60c,4t** for the **KMP_PLACE_THREADS** environment variable. Note that, threads are spread across physical cores 1 through 60 only to support our exploration of the 61^{st} core.

- *Selective*: This affinity pattern binds the control thread to either logical core 240, 241, 242, 243, or 0 depending upon the values of nt and *nThreads*. Task execution threads are allowed run anywhere among the logical cores preceding the control thread.

To facilitate exploration of the 61^{st} core with the *Compact*, *Scatter*, and *Selective* affinity patterns, multiple values of *nThreads* are used to increment the number of logical cores used on the 61^{st} core from 0 to nt. For example, a run configuration featuring *nThreads* = 180 and nt = 3 uses 0 logical cores on the 61^{st} core. This in mind, the control thread is bound to the last logical core

used by a given pattern. For example, a run configuration featuring *nThreads* = 180 and nt = 3 binds the control thread to logical core 240.

Thread Placement with PThreads

Alluded to in the prior subsection, OpenMP users are provided with environment variables to specify thread placement. Such environment variables include **KMP_AFFINITY** and **KMP_PLACE_THREADS**. To learn more about these environment variables, please refer to the **OpenMP Support** section in Intel's *User and Reference Guide* for the corresponding compiler.

In contrast to OpenMP, PThreads requires that developers manually implement affinity patterns. This is attainable using the **CPU_ZERO()**, **CPU_SET()**, and **sched_setaffinity()** functions found within the **sched.h** header file. Given these functions, the **mask** parameter in **sched_setaffinity()** is of key interest. This bit mask determines which logical core(s) a thread is eligible to run on. Figure 1.6 demonstrates how to bind a thread (identified by *pid*) to a single logical core (identified by *logCore*). Referring to the version of Uintah used for this chapter, our use of code in Figure 1.6 can be found in **set_affinity()** in **Thread_pthreads.cc**.

```
cpu_set_t mask;
unsigned int len = sizeof( mask );
CPU_ZERO( &mask );
CPU_SET( logCore, &mask );
sched_setaffinity( pid, len, &mask );
```

Figure 1.6: Specifying one-to-one thread binding

Note however, Figure 1.6 is insufficient for allowing a thread to run among a subset of logical cores. To accomplish this, **CPU_SET()** may be used repeatedly to add additional logical cores to the bit mask. Figure 1.7 demonstrates how to allow a thread to run among a subset of logical cores (specifically, anywhere among logical cores 1 through *nThreads*-1). Referring to the version of Uintah used for this chapter, our use of code in Figure 1.7 can be found in **set_affinityMICMainOnly()** in **Thread_pthreads.cc**.

```
cpu_set_t mask;
unsigned int len = sizeof( mask );
CPU_ZERO( &mask );
for ( int logCore = 1; logCore < nThreads; logCore++ ) {
  CPU_SET( logCore, &mask );
}
sched_setaffinity( pid, len, &mask );
```

Figure 1.7: Specifying one-to-many thread binding

> ♪ **NOTE**
>
> If not explicitly changed, a user-defined **mask** persists for subsequently launched threads. Attention to this detail is critical when implementing affinity patterns with selective thread binding. Such cases require a combination of the methods described in Figure 1.6 and Figure 1.7.

Implementing Scatter Affinity with PThreads

The scatter affinity pattern requires more effort to implement than other patterns we have discussed. For this reason, Figure 1.8 demonstrates how to implement the scatter affinity pattern with PThreads. This example assumes that each thread is uniquely identified by a *threadID* and calls **scatterAffinity()** to denote which logical core it is eligible to run on. Note that, Figure 1.8 supports values of 0 through 243 for *threadID*, where *threadID* = 0 is mapped to logical core 0.

Referring to the version of Uintah used for this chapter, our use of code in Figure 1.8 can be found in **run()** in **ThreadedMPIScheduler.cc**. Note that, our implementation differs slightly from Figure 1.8. Specifically, our implementation handles the 61^{st} core as a special case and spreads threads across physical cores 1 through 60 only. This has been done to support our exploration of the 61^{st} core.

```
void scatterAffinity( int threadID ) {

  int scatterPhysCores    = 61;
  int logCoresPerPhysCore = 4;
  int logCoreIndex        = 0;
  int physCoreIndex       = 0;
  int overallIndex        = 0;

  // Determine whether the thread will be bound to the 1st, 2nd,
  // 3rd, or 4th logical core in a given physical core
  logCoreIndex = floor(( threadID-1 ) / scatterPhysCores ) + 1;

  // Determine which physical core the thread will be bound to
  physCoreIndex = ( threadID - (( logCoreIndex-1 ) * scatterPhysCores ));

  // Determine the specific logical core the thread will be bound to
  overallIndex = logCoreIndex + ( physCoreIndex-1 ) * logCoresPerPhysCore;

  // Bind the thread to its corresponding logical core
  cpu_set_t mask;
  unsigned int len = sizeof( mask );
  CPU_ZERO( &mask );
  CPU_SET( overallIndex, &mask );
  sched_setaffinity( 0, len, &mask );
}
```

Figure 1.8: PThreads-based implementation of the scatter affinity pattern

Experimental Discussion

This section describes our experimental setup and concludes with discussion of our experimental results.

Machine Configuration

These experiments were performed on a single-node machine using one MPI process and double-precision floating point numbers.

Host-side simulations were launched with 32 threads distributed among 16 physical cores. This was accomplished using two Intel Xeon E5-2680 processors in a dual-socket configuration.

Coprocessor-side simulations were launched with as many as 244 threads distributed among 61 physical cores. This was accomplished using one 16 GB Intel Xeon Phi 7110P coprocessor.

Simulation Configuration

Below are key parameters explored on the coprocessor-side:

- 3 physical core usage levels (2, 3, and 4 hardware threads per physical core)
  - For 2 hardware threads per physical core, 3 thread counts were used to allot 0-2 threads for the 61^{st} core (120-122 threads).
  - For 3 hardware threads per physical core, 4 thread counts were used to allot 0-3 threads for the 61^{st} core (180-183 threads).
  - For 4 hardware threads per physical core, 5 thread counts were used to allot 0-4 threads for the 61^{st} core (240-44 threads).
- 4 affinity patterns (*Compact*, *None*, *Scatter*, and *Selective* affinity)
- 4 mesh patch counts (facilitating ratios of 1, 2, 4, and 8 patches per thread)

Below are notes regarding simulation configuration:

- Simulation meshes are decomposed into mesh patches consisting of individual cells.
- Tasks are executed by threads, which are bound to logical cores.
- Tasks reside on mesh patches, which are computed serially using a single thread.
- Different threads may be used to compute tasks resident to a particular mesh patch.
- Tasks are assigned to idle threads without regard to spatial locality of the mesh patch data that they access.

- Simulations were performed using a single-level 128^3 simulation mesh as this was the largest supported by the coprocessor.
- Radiation modeling calculations were performed over 10 consecutive timesteps.
- At each compute timestep, the simulation mesh was sampled using 100 rays per cell.
- Host-side simulations explored the use of 32 threads with the aforementioned affinity patterns and mesh patch counts.

Coprocessor-Side Results

Figure 1.9 through Figure 1.12 visualize results from the 192 simulations performed on the coprocessor-side. Below are notes regarding coprocessor-side results:

- Marks correspond to the average elapsed execution time per compute timestep (in seconds).
- Threads per physical core (TPPC) corresponds to the number of hardware threads utilized per physical core.
- Reserved core usage (RCU) corresponds to number of hardware threads utilized on the reserved core. For clarity, each value of RCU has been enumerated below:
  - RCU = 0 corresponds to use of 120, 180, and 240 threads facilitating 2, 3, and 4 hardware threads per physical core, respectively.
  - RCU = 1 corresponds to use of 121, 181, and 241 threads facilitating 2, 3, and 4 hardware threads per physical core, respectively.
  - RCU = 2 corresponds to use of 122, 182, and 242 threads facilitating 2, 3, and 4 hardware threads per physical core, respectively.
  - RCU = 3 corresponds to use of 183 and 243 threads facilitating 3 and 4 hardware threads per physical core, respectively.
  - RCU = 4 corresponds to use of 244 threads facilitating 4 hardware threads per physical core.
- Patches per thread (PPT) corresponds to the ratio of mesh patches to threads. Note that, each thread is not guaranteed to compute this number of mesh patches.
- Over 10 identical coprocessor-side simulations, there existed not more than a 4.29% difference in performance between two identical runs.
- At 2, 3, and 4 hardware threads per physical core, there existed 30.14%, 42.60%, and 149.33% differences in performance, respectively, between the fastest and slowest run configurations.

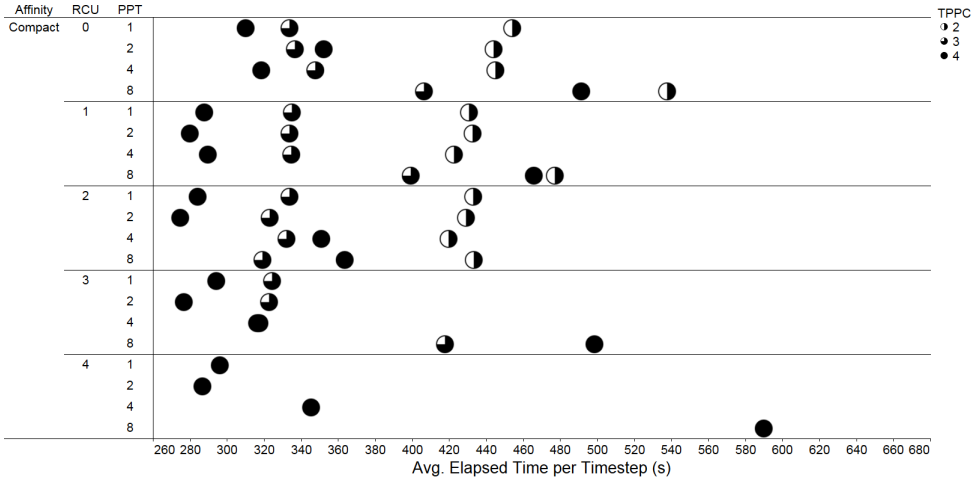

Figure 1.9: Coprocessor-side results for the *Compact* affinity pattern

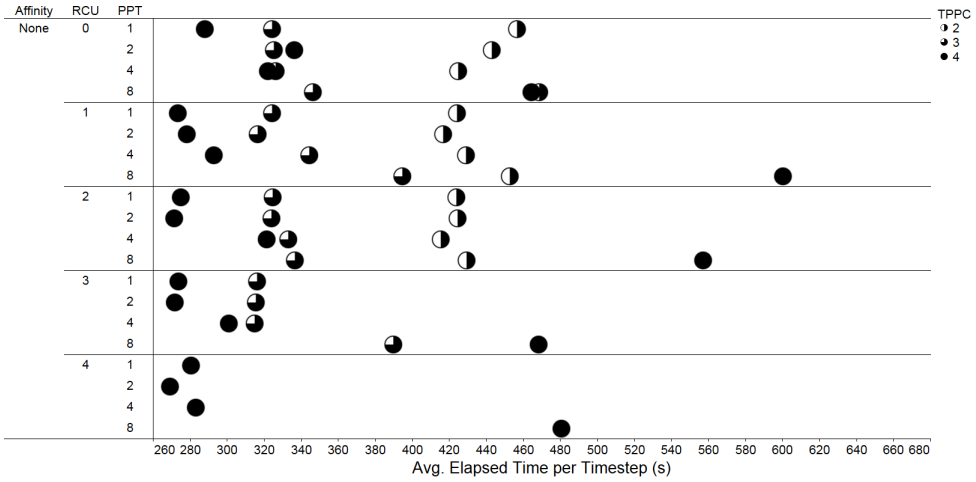

Figure 1.10: Coprocessor-side results for the *None* affinity pattern

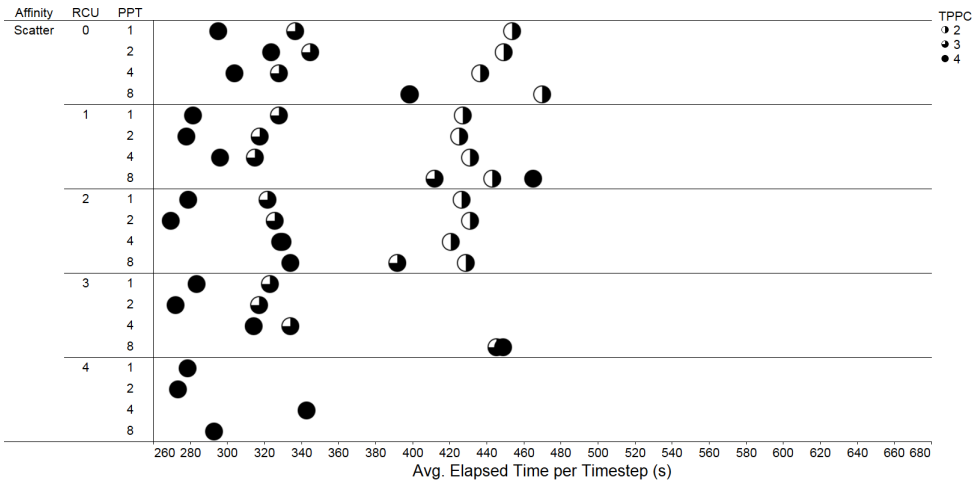

Figure 1.11: Coprocessor-side results for the *Scatter* affinity pattern

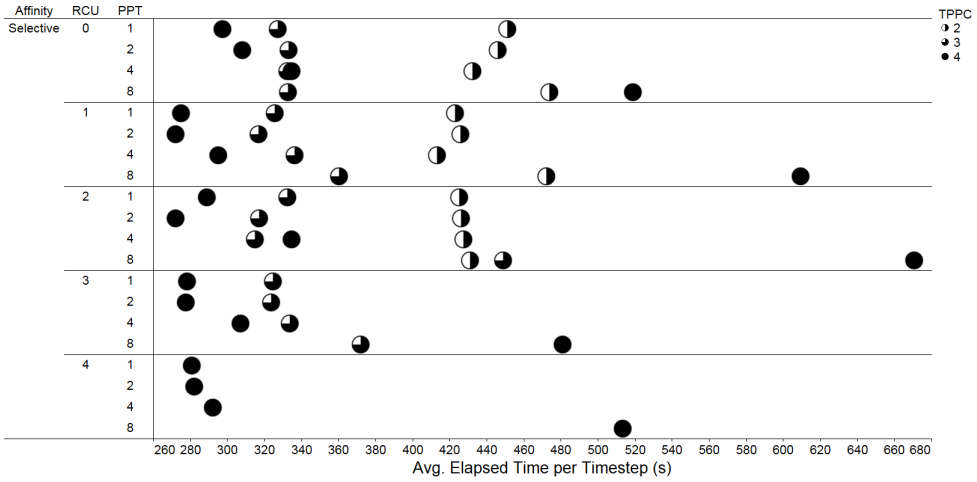

Figure 1.12: Coprocessor-side results for the *Selective* affinity pattern

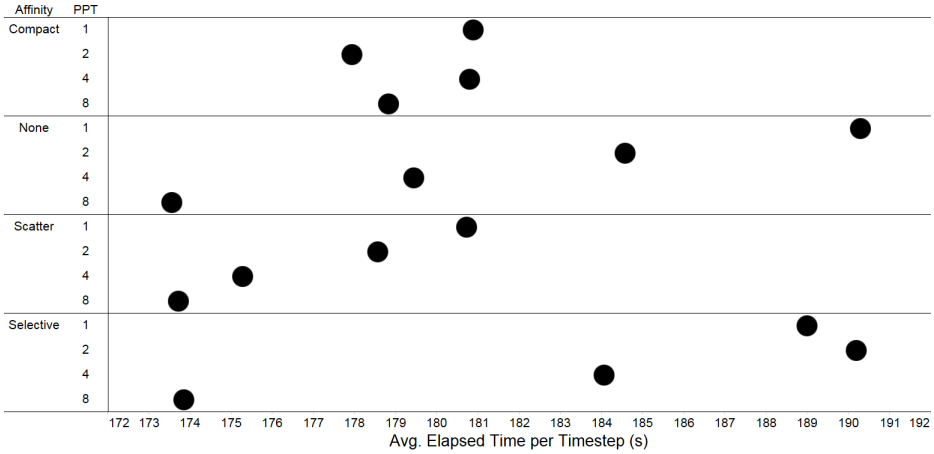

Figure 1.13: Host-side results

Host-Side Results

Figure 1.13 visualizes results from the 16 simulations performed on the host-side. Below are notes regarding host-side results:

- Marks correspond to the average elapsed execution time per compute timestep (in seconds).
- Patches per thread (PPT) corresponds to the ratio of mesh patches to threads. Note that, each thread is not guaranteed to compute this number of mesh patches.
- Over 10 identical host-side simulations, there existed not more than a 3.35% difference in performance between two identical runs.
- There existed a 10.14% difference in performance between the fastest and slowest run configurations.

Further Analysis

Addressing comparisons between architectures first, the two Xeon processors outperformed the single Xeon Phi coprocessor. Specifically, there existed a 39.43% difference in performance between the fastest run configurations for each architecture. Regarding accuracy, simulation results computed by each architecture were identical to one another up to a relative tolerance of 1E-15 digits.

Given this has been a naive port of our CPU-based algorithm, these results are encouraging as they leave ample opportunity to shift performance in favor

of the coprocessor. Having not yet adequately pursued such optimizations, effective memory management and vectorization are believed to be the factors attributing to these differences. Supporting this conclusion, version 15.0 compiler optimization reports and experimentation with simpler vectorization approaches (e.g. SIMD directives) suggest that little, if any, vectorization is being achieved. Further, predominantly 100% core usage during compute timesteps suggests that thread-level parallelism is sufficient.

Turning to observations, performance disparities among coprocessor-based results deserve attention. As more hardware threads per physical core we utilized, the difference in performance between fastest and slowest run configurations increased. This is likely attributed to the sharing of the 512 KB per core L2 cache among four hardware threads. Though it offered better run times, use of more hardware threads per physical core further divided the amount of L2 cache available to a given thread. This resulted in increased sensitivity to simulation mesh decomposition.

Moving forward, the overarching takeaway from these native execution-based experiments is that no one thread placement strategy dominated performance. For similar algorithms, this suggests that time may be best spent by first pursuing more favorable areas of optimization.

Returning to the question motivating this work, our fastest run configurations utilized the 61^{st} core. Further, performance was not profoundly impacted when explicitly oversubscribing the coprocessor operating system thread. For similar algorithms, this suggests that use of the 61^{st} core may be both forgiving and capable of offering modest performance improvements.

Summary

These experiments have helped establish valuable baselines for our future efforts addressing both single-node performance and scalability. Perhaps more important, they have also provided valuable insight regarding potential challenges and areas to address as we strive to achieve performance with the Xeon Phi. When considering these results, it is important to remember that we have examined native execution exclusively. When operating in offload mode, Intel guidance is to refrain from using the reserved core as it actively supports offloading.

Acknowledgements

These research efforts have been supported by the National Nuclear Security Administration's PSAAP II project. This work utilized equipment donations to the University of Utah's Intel Parallel Computing Center (IPCC) at the SCI Institute. We would also like to thank Aaron Knoll, IPCC Principal Investigator, for his assistance along the way.

For More Information

To learn more about the Uintah Computational Framework, please refer to the link below:

- `http://www.uintah.utah.edu/`

To explore Uintah and C-SAFE-related publications, please refer to the link below:

- `http://www.uintah.utah.edu/pubs/pubs.html`

To learn more about radiative heat transfer and the discrete ordinates method-based radiation model, please refer to the publication referenced below:

- Spinti, J.P., Thornock, J.N., Eddings, E.G., Smith, P.J., and Sarofim, A.F. "Heat transfer to objects in pool fires." In *Transport Phenomena in Fires*, WIT Press, Southampton, UK (2008).

To learn more about the reverse Monte-Carlo ray tracing-based radiation model, please refer to the publication referenced below:

- Humphrey, A., Meng, Q., Berzins, M., and Harman, T. "Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system." In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment (XSEDE12)*. ACM, 2012.

To download the version of Uintah used for this chapter, please refer to the link below:

- `http://lotsofcores.com/downloads`

To learn more about installing and using the Uintah Computational Framework, please refer to the link below:

- `http://uintah-build.sci.utah.edu/trac/wiki`