Analysis and Optimization of a Molecular Dynamics Code using PAPI and the Vampir Toolchain

Thomas William
ZIH/VDR
TU Dresden
Dresden, Germany
williath@iu.edu

Donald K. Berry
Research Technologies
Indiana University
Bloomington, Indiana
dkberry@iu.edu

Robert Henschel
Research Technologies
Indiana University
Bloomington, Indiana
henschel@iu.edu

***Abstract*—A highly diverse molecular dynamics program for the study of dense matter in white dwarfs and neutron stars was ported and run on a Cray XT5m using MPI, OpenMP and hybrid parallelization. The ultimate goal was to find the best configuration of available code blocks, compiler flags and runtime parameters for the given architecture. The serial code analysis provided the best candidates for parallel parameter sweeps using different MPI/OpenMP settings. Using PAPI counters and applying the Vampir toolchain a thorough analysis of the performance behavior was done. This step led to changes in the OpenMP part of the code yielding higher parallel efficiency to be exploited on machines providing larger core counts. The work was done in a collaboration between PTI (Indiana University) and ZIH (Technische Universität Dresden) on hardware provided by the NSF funded FutureGrid project.**

***Keywords*-molecular dynamics; tracing; MPI; OpenMP;**

I. Introduction

The performance analysis of applications that have been in use for many years always pose special challenges as these codes tend to include many tweaks that have been implemented for special purposes and that obfuscate the view on low level problems. The MD code used by C. J. Horowitz and his group is no exception in that. It consists of several versions of the same semantics implemented in various ways. For example to make use of new language constructs available in newer Fortran versions. There is also a version for the MDGRAPE-2 boards available, a hardware especially designed for molecular dynamics simulations (see appendix of [1] for further details). Together with a wide range of compile time flags that influence the arithmetics of the simulation, this creates several hundred possible application-runs to look at.

In this paper we present our approach on how to conduct such an analysis, starting with the serial version of the code in section III, then using PAPI counters in section IV on page 4 to gain more insight which led to the inevitable look at the source code itself (section V). After that, we applied the Vampir toolchain in section VII to conduct an analysis of the OpenMP, MPI and hybrid version of the code.

II. Background Information

MD was written to simulate certain physical properties of neutron stars and white dwarfs (see [2] and [3]). Over the years the code has been extended and rewritten multiple times to include different types of interactions (nucleon, pure-ion, or ion-mixture). There are several different implementations of the same semantics, for instance the same loop was reimplemented using the Fortran array syntax available since Fortran 95. Additionally, special features have been written over the years such as a cut-off sphere to restrict interactions to a subset of relatively nearby nucleons/ions. There was also an attempt to reorder the do-loop in the ion interaction routine to use blocking techniques. The force calculation routine has to be chosen at the build stage. There are two sections of code that each have two or more variations. One section is labelled `A` and the other `B`, and the variations are numbered. The block size `NBS` has to be set as well. All this has to be specified in the make command:

```
make MDEF=XRay md_mpi BLKA=A0 BLKB=B2 NBS="NBSX=32"
```

MD can be compiled as a serial, OpenMP, MPI or MPI+OpenMP program. The variants have different names, (`md, md_omp, md_mpi, md_mpi_omp`) and are selected during compile time as can be seen above.

The program is a classical molecular dynamics simulation of dense nuclear matter consisting of either fully ionized atoms or free neutrons and protons. Its main targets are studies of the dense matter in white dwarf and neutron stars. Matter in these object is of such high temperature that they may be treated classically, although distances between particles in the system are in the order of femtometers. At such high temperatures, the quantum de Broglie wavelengths of the particles are much shorter than inter-particle separations (see [4]). This justifies treating the interaction potentials between particles as classical two-particle central potentials.

The electronic structure of white dwarf and neutron star matter is much simpler than in terrestrial matter. Atoms are fully ionized and the electrons form a fully degenerated

Fermi gas. The complex electron orbital structure of ordinary matter no longer exists. Thus electrons do not need to be modeled explicitly. Their only effect is to provide an exponential screening of the ordinary Coulomb interaction between ions, with a screening length λ.

A. MD Implementation Details

The interactions between particles are central forces between point particles. This is a rather simple approach compared to two-, three-, and four-particle bonded interactions in molecular systems. There is no complicated particle geometry or orientation. That way, to calculate the total force on an i-particle one simply adds the forces due to all j-particles. This can be done for all i-particles in a doubly nested ij-loop (see figure 1).

For ordinary molecules, the non-bonded interactions are of such short range that one can impose a cut-off radius small enough that each atom interacts with only a few others (10x-100x). In the systems simulated here, each ion interacts with practically all others in the system. This is a drawback, since most molecular systems have a short-range cut-off. Although the Coulomb interaction is exponentially screened with screening length λ, it is still a long range interaction, but not an infinite-like unscreened Coulomb interaction (see [3] and [4]), and each ion interacts with thousands of others.

In fact, early versions of MD did not have any cut-off implemented in the interaction routines as interactions of all particle pairs were computed. This certainly simplified the programming, making the algorithm just the simple particle-particle (PP) algorithm. The algorithm had a time complexity to calculate all the forces of $O(N^2)$. Experiments showed that a cut-off radius could be introduced, but it is still rather large as other physics constraints set the simulation box size to a size that almost exactly circumscribes the cut-off sphere. Thus MD version 6.1.0 used here still includes the PP algorithm.

The structure of the code is simple. It first reads in a parameter file `runmd.in` and an initial particle configuration file `md.in`. The parameter file is a Fortran name list, which defines parameters in a keyword=value format. The `md.in` file is an unformatted file whose first record is a simulation time stamp, and whose second record contains positions and velocities.

After reading the `runmd.in` and `md.in` files the initial configuration and all parameters are set up. The program then calculates the initial set of accelerations, and enters a time-stepping loop afterwards. The loop is actually a triply nested do-loop. Time steps are divided into `ngroup` measurement groups. Each group consists of `ntot` measurements. Measurements are done every `nind` time steps. Thus the code has the basic structure shown in figure 1. The forces are calculated in the newton module in a pair of nested do-loops. The outer loop iterates over target particles, the inner loop over source particles. The targets are assigned to MPI processes in a round-robin fashion. Within each MPI process, the work is shared among OpenMP threads. Profiling shows that most of the time is pent in the subroutines called in the newton module. The function newton is different depending on the type of particles acted upon. These can be nucleon-nucleon (where a nucleon is either a proton or neutron) for simulations of nucleons, pure-ion (where a single ion species is simulated), or mixed ion interactions. The code takes advantage of Newton's third law, so that only $n * (n-1)/2$ interactions have to be calculated. As stated above, these particle-particle-interactions have been implemented in a multitude of ways over the years and are located in different files PP01, PP02 and PP03. PP01 is the original implementation with no splitting into the Ax, Bx or NBS blocks mentioned above and is used as a baseline when benchmarking the other implementations. PP02 implements the versions in use by the physicists today and features 3 different implementations for the Ax block, 3 implementations for the Bx block and no manual blocking (NBS). PP03 has 3 Ax blocks, 8 Bx blocks and can be blocked using the NBS value that has to be given at compile time together with the A and B values. The code blocks are selectable using preprocessor macros. The blocks B0 and B1 for instance only differ in that B1 uses the newer array syntax available since Fortran 90. Whether or not these version yield the same performance will be determined in section III.

```
      do 100 ig=1,ngroup
         initialize group ig statistics
         do 40 j=1,ntot
            do i=1,nind
               !computes forces
               !updates x and v
               call newton
            enddo
            call vtot
         enddo
         compute group ig statistics
 40      continue
100   continue
```

Figure 1. Simplified version of the main loop

B. Cray XT5m™

All the measurements are done on XRay, a Cray XT5m provided by the FutureGrid project NSF grant 0910812 [5]. The XT5m is a 2D mesh of nodes. Each node has two sockets each having four cores of type AMD Opteron 23 "Shanghai" (45 mm). There are 84 compute nodes with a total of 672 available cores running at 2.4 GHz.

The batch scheduler interfaces with the Cray resource scheduler (APLS). When there is a job submission, ALPS calculates the available resources and makes the reservation upon request from the batch scheduler. ALPS is a `gang`

scheduler, this means that there is only one job per node allowed. A requests like:

```
aprun -n 1 ./md.exe
```

will leave the other seven cores on the scheduled node empty. The following request will be schedule to the next node regardless of whether it could fill up the empty cores on the scheduled node or not. This is beneficial while running the serial jobs as they are all scheduled to individual nodes automatically and thus do not interfere with each other.

XRay has a PAPI module xt-papi/3.6.2.2 that is provided by Cray. The compiler used is pgi/9.0.4, also provided as a module. Additionally the appropriate XT PE driver xtpe-shanghai adds "-tp shanghai-64" to the compiler flags.

III. Serial Analysis

In the first step, we look at the performance of the serial version of MD. To narrow down the number of possible candidates for a parallel analysis, we compare all the available code versions, test different compiler flags and look at the weak scaling capabilities using main input parameters such as the particle count. The input dataset we will use consists of a mix of ions (oxygen, neon, carbon). This means that the functions dealing with pure-ion or nucleon-nucleon interactions will not be part of the analysis, which is also true for the OpenMP and MPI analysis. For a small input dataset of only 5k particles (5120), we will simulate 1280 carbon ions, 3740 oxygen ions and 100 neon ions. Higher particle counts are scaled accordingly. The runtime parameters for the algorithm depicted in figure 1 are as follows:

```
!Parameters:
   sim_type ='ion-mixture', !simulation type
   tstart   =      0.00,    !start time
   dt       =     25.00,    !time step (fm/c)
!Warmup:
   nwgroup  =        2,     !groups
   nwsteps  =       50,     !steps per group
!Measurement:
   ngroup   =        2,     !groups
   ntot     =        2,     !per group
   nind     =       25,     !steps between
   tnormalize =     50,     !temp normal.
   ncom     =       50,     !center-of-mass
                            !motion cancel.
```

While such a 5k particle measurement only takes 5 minutes, running the same simulation with 27k particles takes roughly an hour. Scientifically useful data can be obtained with 55k particles or more. This takes 10h hours on XRay to complete. The default optimization flag used is -O3. We also measured -O2 as this is the compiler-default. To make use of the SSE registers -fastsse is used. On fairly recent PGI compilers this equals -fast, which in turn includes the following flags:

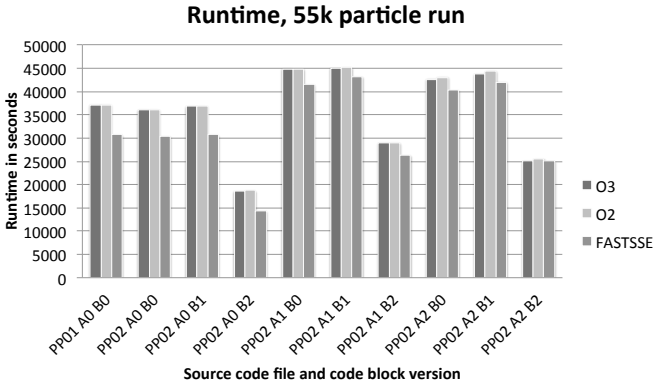

Figure 3. Overview of the runtimes for the code-blocks combinations in the PP01 and PP02 source file in scientific use right now for an input dataset of 55k particles. NBS was set to 0.

```
-O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline
-Mvect=sse -Mscalarsse -Mcache_align -Mflushz
```

Figure 2 shows an overview over all combinations. Blocking with NBS was measured for a blocking factor of 2, 8, 16, 32, 64, 128, and 256. Measurements are sorted starting on the left from PP01 and A0+B0 code-block in lexicographical order. The x-axis is runtime in seconds, so smaller is better. The graph shows little difference between -O2 and -O3, but the -fastsse seems to have greater impact when used in combination with blocking. The blocking is mapped directly to the SSE registers. A blocking factor higher than 8 does not yield better performance. Only looking at the PP03 versions it seems that the A0 versions are faster than the A1 and A2 blocks when compared to its corresponding Bx block. Additionally, there seems to be a huge gain in using B3 an higher. Nonetheless the shortest execution time is achieved when using a combination of the PP02 file. All basic principles are also present in the PP02 file and PP03 consists of several attempts to re-implement the same semantics present in PP02 using more modern programming techniques of Fortran 95. As these seem to have no significant impact and as the fastest version is present in PP02, we will concentrate the analysis on PP02. Figure 3 shows that the A0 block seems to be more efficient than A1 or A2. The -O3 flag only affects the A2 block differently than the -O2 flag. The annotated assembler in figure 4 and 5 shows that the SSE registers are already used by default although for -O2 and -O3 they are merely used as a fast type of "normal" 64 bit registers without making use of its 128 bits. This is also true for the -fastsse flag as the high quadword of the registers remains unchanged and only directives using 64 bit operands are used. The "sd" in mul, add, sub, and sqrt means scalar double-precision and only uses the first 64 bit of a register. The "lpd" stands for low packed double precision and also means that the high quadword of the register remain unchanged by the operation.

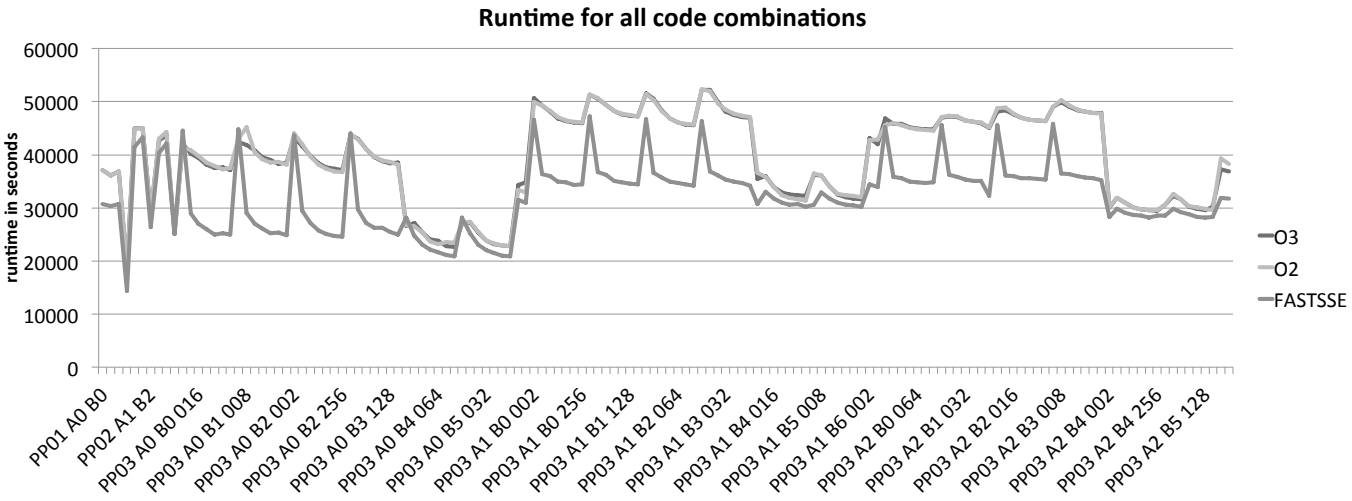

Figure 2. Overview of the runtimes for all code-block combinations with an ion-mix input dataset of 55k particles. The naming scheme for the measurements is "source-code-file A-block B-block blockingfactor"

```
##              do 90 j=i+1,n-1
##              !------ Block A ------
## #if  defined(A0)
##                 r2=0.0d0
   movsd      %xmm2, %xmm1
   movq       %r12, %rdx
   movq       %r15, %rcx
   movl       $8, %eax
   .align       16
.LB2_555:
##   lineno:  138
```

Figure 4. Code example showing that SSE registers are used when `-O3` is used

```
##          do 90 j=i+1,n-1
##          !------ Block A ------
## #if  defined(A0)
##              r2=0.0d0
##              do k=1,3
##                 xx(k)=x(k,i)-x(k,j)
   movlpd     .BSS2+48(%rip),%xmm2
   movlpd     .C2_291(%rip),%xmm0
   mulsd      %xmm2,%xmm2
   addsd      %xmm1,%xmm2
   movlpd     %xmm2,344(%rsp)
   sqrtsd     %xmm2,%xmm2
   movlpd     %xmm2,448(%rsp)
   mulsd      md_globals_10_+120(%rip),%xmm2
   subsd      %xmm2,%xmm0
   .p2align    4,,1
```

Figure 5. With `-fast` more heavy usage of SSE registers can be observed

The `-fastsse` flag yields slightly better results with the exception of B2. This effect is further explained in section V. The largest impact on runtime seems to origin from the usage of the B2 block. In case of `-fastsse`, this cuts down execution time by half from 30.000 seconds to less then 15.000. Trying to understand why the code blocks have these different performance characteristics we next tried to apply the VampirTrace framework to the code. As MD is MPI and OpenMP parallel, VampirTrace automatically used its internal source-to-source compiler Opari to instrument the OpenMP pragmas. This boosted the runtime from 40 seconds to over 4000 seconds for 1k particles. While the traces were of no direct value as they did not resemble the codes real behavior due to the large overhead it was an indicator that OpenMP usage might have performance issues as the overhead induced by VampirTrace is normally in the range of single digit percentage. See [6] for a thorough overhead analysis using the SPEC MPI 1.0 suite as example.

In order to understand the differences in runtime of the code blocks we manually implement PAPI counters as a next step.

IV. PAPI Measurements

The code is altered to allow for a PAPI counter to be set during compile time. Inside the newton subroutine (see figure 1) the PAPI counter is measured for each call to the acceleration subroutine. The acceleration subroutine consists of a case statement that switches between the nucleon, pure-ion, and ion-mixture simulation type. By wrapping the subroutine we get a PAPI-value at each simulation time step $t + \delta t$. The values are written to a file and summed up at the end of the simulation run. Creating different binaries for the different PAPI counters allows us to run all the binaries with the same input dataset in one batch-job. This

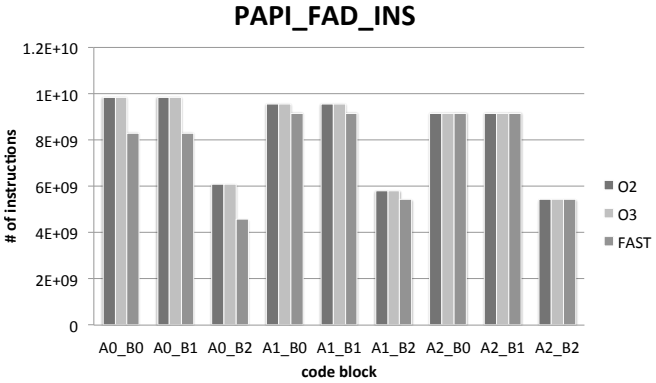

Figure 6. Floating point add instructions

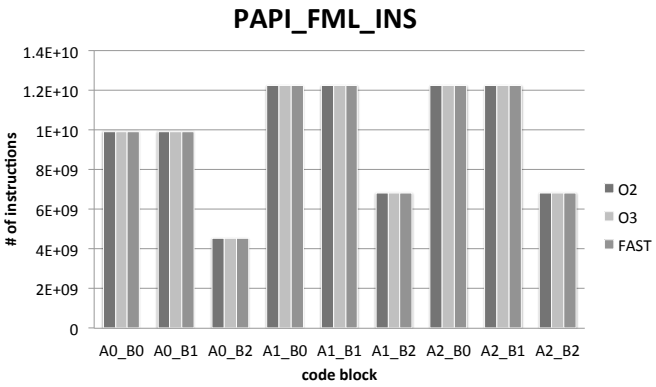

Figure 7. Floating point mult instructions

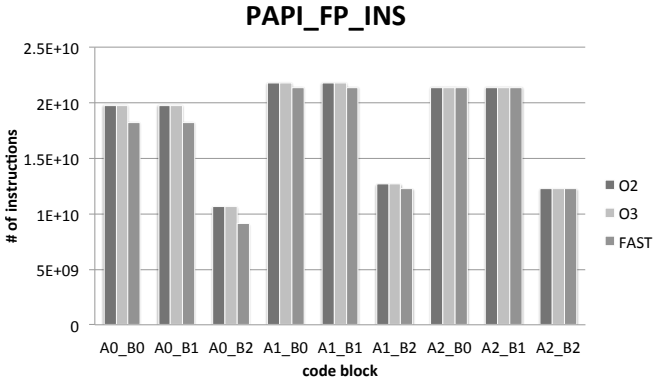

Figure 8. Number of floating point instructions for a 27k particle run

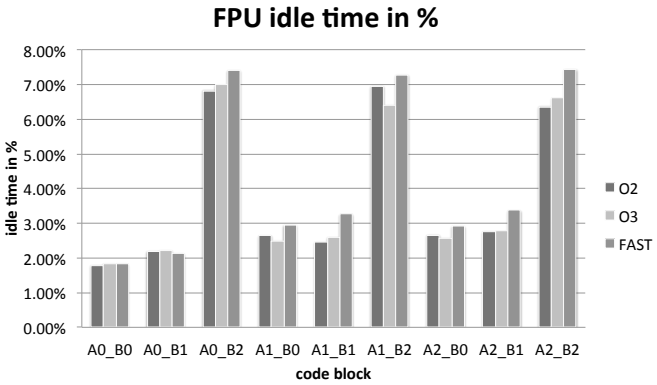

Figure 9. FPU idle times in percent of PAPI measured total cycles

way the environment is completely identical for all measured counters and thus the values can be compared.

All PAPI counters are measured using 27k particles to cut down on runtime. 18 different counters were measured for all 9 PP02 blocks using `-O2`, `-O3`, and `-fast`. As it turns out, for the measurements we did on XRay and judging by the raw numbers read from the counters, the floating point counters seem relate as follows:

```
PAPI_FAD_INS+PAPI_FML_INS=PAPI_FP_INS=PAPI_FP_OPS
```

Figure 8 shows the number of floating point instructions for each code block. For all compiler flags, B0 and B1 always have exactly the same value. Despite the fact that all blocks calculate the same result, B2 seems to need only 54%(A0), 57%(A2), 58%(A1) of the instructions compared to B0/B1. In case of A2 the values for the compiler flags are completely identical regardles of the used compiler flag. Figures 6 and 7 show that the difference between the compiler flags origins solely from the add instructions. At the same time the FPU is idle 2% of the total cycles for B0/B1 but 7% for B2 as figure 9 shows. The number of branch instructions of B2 is lower compared to the other B blocks (figure 10), but the absolute number of miss predictions is higher (figure 11). Looking at the branch miss prediction rate (PAPI_BR_MSP / PAPI_BR_INS) shows a nearly twice as high value for B2 compared to B0 or B1. The L1 hit ratio is also marginally better for B0/B1 (99.95%) compared to B2 (99.85%). Normally, more branch miss predictions, lower cache hit ratio, and more FPU idle time indicate a lower performance but here this is connected to less floating point instructions and a lower total runtime. To explain this behavior a look into the source code is inevitable.

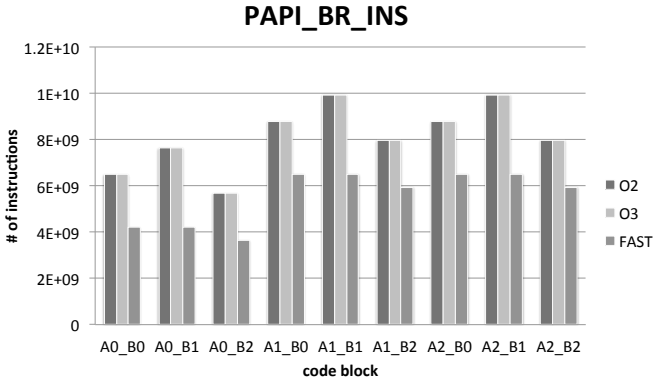

Figure 10. Branch Instructions

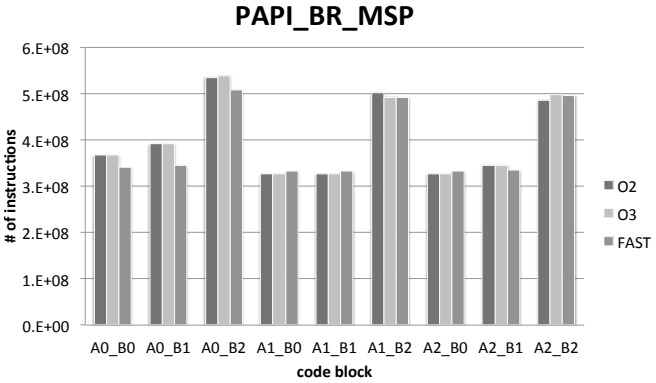

Figure 11. Branch miss predictions

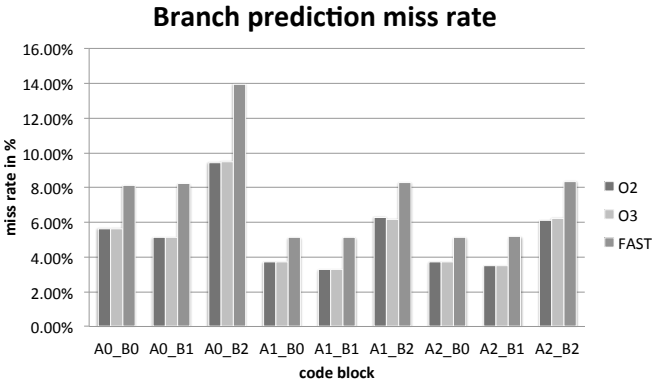

Figure 12. Branch miss prediction rate in percent

V. Source Code Analysis

With the knowledge gained from using the PAPI counters it is now possible to relate the performance to the code blocks shown in figure 13 and 14.

Each of the blocks does something different compared to the other that can be used to categorize the sections. A0 uses branching with an if statement whereas A1 uses more arithmetic calculations without the "if". A2 omits the "do" by using the array syntax. This has the positive side effect that the variable `r2` does not have to be set to zero initially. B0 uses a do-loop compared to B1 that does exactly the same, but uses the array syntax. B2 is different in that it features a "if" that checks whether the target particle is inside the cut-off sphere or skips the calculation otherwise.

Branching with an if statement in A0 is faster than doing all the arithmetics like A1 and A2 do (see figure 3). The additional assignment of `r2` in front of the small loop is the reason for A1 of being 3-9% slower than A2, whose only other difference is the usage of a different syntax. That the different syntax leads to nearly the same performance can be observed by looking at B0 and B1. These blocks only differ in that B0 uses a loop whereas B1 uses the array syntax. After accounting for the measuring inaccuracy, B0 is less than 1% faster than B1. The A2 block produces slightly

```
#if defined(A0)
  r2=0.0d0
  do k=1,3
     xx(k)=x(k,i)-x(k,j)
     if(xx(k).gt.+halfl(k)) xx(k)=xx(k)-xl(k)
     if(xx(k).lt.-halfl(k)) xx(k)=xx(k)+xl(k)
     r2=r2+xx(k)*xx(k)
  enddo
#elif defined(A1)
  r2=0.0d0
  do k=1,3
     xx(k)=x(k,i)-x(k,j)
     xx(k)=xx(k)-aint(xx(k)*halfli(k))*xl(k)
     r2=r2+xx(k)*xx(k)
  enddo
#elif defined(A2)
  xx(:) = x(:,i)-x(:,j)
  xx=xx-aint(xx*halfli)*xl
  r2=xx(1)*xx(1)+xx(2)*xx(2)+xx(3)*xx(3)
#else
```

Figure 13. Block A of the PP02 file for the ion-mix interactions

```
#if defined(B0)
  r=sqrt(r2)
  fc = exp(-xmuc*r)*(1./r+xmuc)/r2
  do k=1,3
     fi(k)    = fi(k)   + zii(j)*fc*xx(k)
     fj(k,j)  = fj(k,j) - zii(i)*fc*xx(k)
  enddo
#elif defined(B1)
  r=sqrt(r2)
  fc = exp(-xmuc*r)*(1./r+xmuc)/r2
  fi(:)    = fi(:)   + zii(j)*fc*xx(:)
  fj(:,j)  = fj(:,j) - zii(i)*fc*xx(:)
#elif defined(B2)
  if(r2.le.rcutoff2) then
     r=sqrt(r2)
     fc=exp(-xmuc*r)*(1./r+xmuc)/r2
     fi(:)    = fi(:)   + zii(j)*fc*xx(:)
     fj(:,j)  = fj(:,j) - zii(i)*fc*xx(:)
  endif
#else
```

Figure 14. Block B of the PP02 file for the ion-mix interactions

better code with the `-O3` than when compiled with `-O2`. The reason for this is not entirely known. But as the effect is the same for all the three runs that use the A2 block, we guess that the array syntax might allow the compiler to apply additional optimizations not possible otherwise.

By far the largest impact on runtime is achieved by using the cut-off sphere that restricts the number of particles that has to be acted upon. Block B2 uses this cut-off sphere, which is a radius around the source particle. Only for target particles inside the sphere, particle interactions are calculated. Although the additional if statement at the beginning leads to higher branch miss predictions (see figure

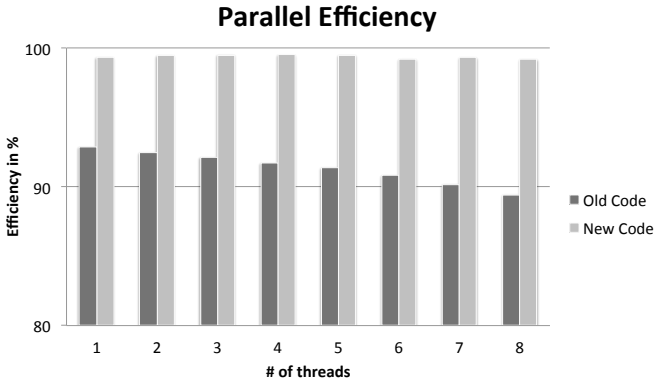

Figure 15. Parallel efficiency of the two code versions running as a pure OpenMP application

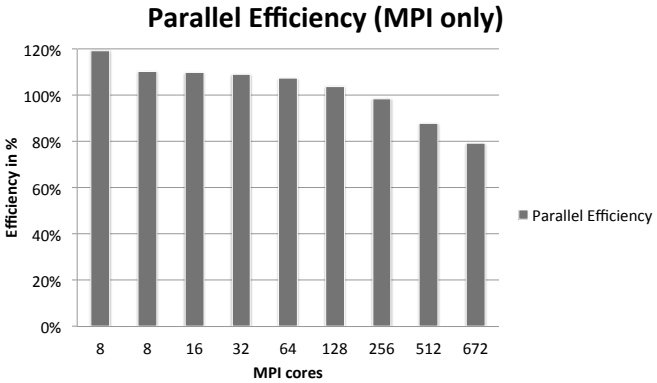

Figure 16. Parallel efficiency of the MPI version

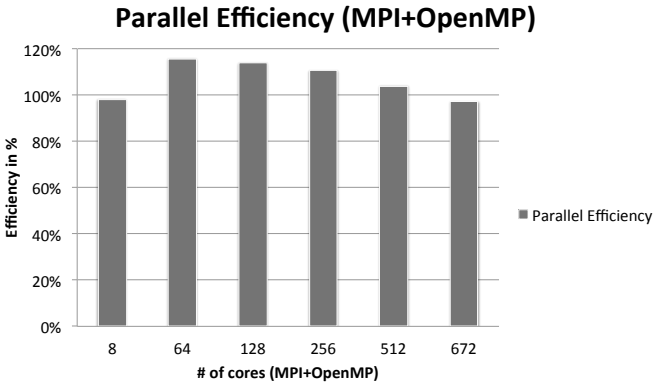

Figure 17. Parallel efficiency of the hybrid version using MPI and OpenMP

11 and 12) and a lower cache hit ratio as pointed out in section IV, the cut-off-sphere used in the B2 block results in lesser particle-particle interactions having to be calculated compared to the B0 and B1 blocks. Comparing the runtime of figure 3 with the number of floating point instructions needed in figure 8 one can see a direct correlation between the two and the impact of the cut-off sphere.

To sum up, in this case, calculation does not beat branching. But that might change for future hardware and different input datasets. Additionally, using a cut-off sphere changes the mathematics of the simulation. If used wisely, the effects will not alter the scientific outcome but computationally the code does something different.

VI. Source Code Optimization

In section III we tried to apply the Vampir toolchain to the code but this made the application run 100 times slower than usual. With the knowledge from the PAPI counters and our analysis of the source code it is now possible to find the reason for this unusual overhead. Running the MPI-only version had only a 3% overhead. So the focus is on the OpenMP pragmas. The trace shows that most of the time spent inside OpenMP is actually spent in thread management instead of computation. Although the reality is distorted by the fact that VampirTrace also uses these functions to create the necessary trace data to log the threads, which further adds to the overhead, this behavior indicates that the workload per thread is to small and performance is impeded.

The new code version differs from the original in that the `omp parallel do` on the j-loop is replaced by an `omp parallel` region around the i-loop, and the j-loop is parallelized explicitly, with no OpenMP directives. This is done by enclosing the entire ij-loop section in an 3rd loop over threads. Each iteration of this outer loop is assigned to an OpenMP thread, which then calculates the j-loop iterations it is responsible for.

Figure 15 shows that even the single thread performance benefits from the change to 3 loops. The serial version of the old code needed 14504 seconds for a 1 Core, 55k particles run. Due to the OpenMP overhead this increased to 15622 seconds running 1 OpenMP thread. With the new version the OpenMP overhead is lower and this now takes 14602 seconds to complete.

While the old version drops below 90% efficiency with 8 threads, the new version starts at 99.3% with a single thread, peaks at 3 threads with 99.5% and achieves 99.2% with 8 threads. This is the maximum number of threads on the Cray XT5m. Newer hardware architectures with more cores like the AMD bulldozer will benefit from this more heavily.

The graph for the parallel efficiency of the MPI-only version, figure 16, has two values for 8 MPI processes. Intra-node MPI is shown first, the second value was measured using 8 nodes with 1 process per node. Up to 128 processes, the efficiency is above one due to cache effects. Using all 672 available cores on XRay running the MPI-only version, efficiency drops below 80%. This behavior can be explained by looking at the traces in the next section.

To maximize the performance on the given hardware, OpenMP and MPI are combined. We start with 8 cores on one node using 4 OpenMP threads and 2 MPI processes to resemble the dual socket quad core hardware. This yields 98.2% which is 1% less than the pure OpenMP version

shown in figure 15. All subsequent measurements were therefore taken using MPI inter-node and OpenMP intra-node. So the 64 core run uses 8 nodes and 672 core run uses all available 84 nodes. Using the full machine and the optimized OpenMP code, an efficiency of 97.2% can be achieved compared to 79.2% for the MPI-only version.

VII. Tracing and Visualization

VampirTrace increases the runtime by ∼7% with the new code instead of a factor of ∼100. This makes a analysis with Vampir feasible. Vampir shows only a part of the trace file of the MPI-only version in figure 18. Only a few of the 672 available processes are shown here. The selected part is approximately 3.3 seconds of the 27 seconds long trace, starting from the middle of the run as shown in the navigation bar at the top right. The function summary on the right shows that most of the time is spent in a function called `accel_ion_mix`, the function which includes the code block shown in figure 13 and 14. It is colored blue. The larger red section on the left, in the top left window, is a `MPI_barrier` used to sync between the two groups. The number of groups (`ngroup`) is a variable set in the runtime parameters, see section III. Each acceleration step is followed by a `MPI_Allreduce`, the small red bars. The communication is marked with the black lines connecting the communications partners with each other. Every `nind` steps, the function `vtot_ion_mix` calculates the average potential energy per ion. This is the purple bar at the right which concludes the 25 iterations that were set in the configuration, see section III. As can be seen from the trace, the all_reduce-call is the only MPI routine used (except the single call to `MPI_barrier`). The energy per particle is distributed using the `MPI_Allreduce` which combines the values from all processes and distributes the result back to all processes. PAPI counters show 400 Mflop/s in floating point performance (red line) as well as a drop to zero (blue line) near where the MPI call is active and a not-instant recovery back to 400 Mflop/s.

This is due to the fact that after each `MPI_Allreduce` call, which does not do any floating point arithmetic, the `accel_ion_mix` subroutine finishes and the code jumps back to the inner loop shown in figure 1 on page 2. Whenever `newton` is called, the coordinates need to get updated and new velocities and acceleration at time $t + \delta t$ have to be calculated. This happens at 200-300 Mflop/s. The stack levels are shown in the process timeline at the lower right of figure 18. There the "gap" between two `accel_ion_mix` calls can be seen.

When the normalization function is called which in turn calls the subroutine `vtot` (purple part at the end) to calculate the average kinetic energy per particle that causes a very high floating point ratio.

According to this, the limiting factor for the MPI version of the code is the `MPI_Allreduce` which is needed to synchronize the processes after each step so that the particle-particle interactions are calculated using updated values. This does not provide room for improvement other than not using global communication but point-to-point messages, or different MPI communicators, to update processes inside the cut-off-sphere. This has not been done so far.

Table I
MPI-only, PP02 A0 B2 run, all processes, accumulated exclusive time per function

	672 cores		8 cores	
function name	time in s	%	time in s	%
accel_ion_mix	13293.4	72.7	11735.0	97.9
MPI_Allreduce	1885.8	10.3	44.1	0.4
sync	939.4	5.1	0.1	0
newton	937.2	5.1	4.9	0.0
MPI_Bcast	807.1	4.4	8.2	0.1
vtot_ion_mix	189.6	1.0	188.2	1.6
MPI_Init	112.8	0.6	0.1	0
MPI_Barrier	107.5	0.6	1.2	0

Vampir can sum up the PAPI counters of all 672 processes and present them as a graph over time aligned to the function calls. For the 672 core MPI-only run, floating point performance is at 400 Mflop/s on average with peaks at 800 Mflop/s, see figure 13 and 14). The optimized OpenMP version reaches 700 Mflop/s on average with peaks up to 1.1 Gflop/s.

Not using the cut-off sphere leads to more calculations per thread/core and thus to a higher value for the floating point counter. As, for the cut-off sphere to work, an additional if statement has to be used. The $\frac{4}{5}$ saved in calculation in the acceleration subroutine lead to a factor 2 in speedup. This correlates to half the floating point instructions needed (compare figure 8 and 3) in the overall simulation run compared to not using a cut-off sphere. Please note that PAPI explicitly states for the `PAPI_FP_OPS` counter to count speculative adds and multiplies and that this counter is variable and higher than theoretical.

Table I shows all functions that need 100 seconds or more in the 672 core full machine run. Compared is a run using 8 nodes and 1 process per node. 15.9% of the runtime is needed for MPI routines compared to only 0.5% for the 8 core run. Especially the time needed for collective operations like the `MPI_Allreduce` will grow faster with a rising number of cores. This indicates that MPI is indeed the limiting factor in scaling the application to larger core counts.

VIII. Conclusion

The analysis of MD found the best serial version. Using PAPI and analyzing the source code revealed that the reason for the A0 B2 block being nearly two times faster than the original code version is due to the fact that using a cut-off sphere, the number of particle-particle interactions is reduced to 19%, leading to a runtime decrease although

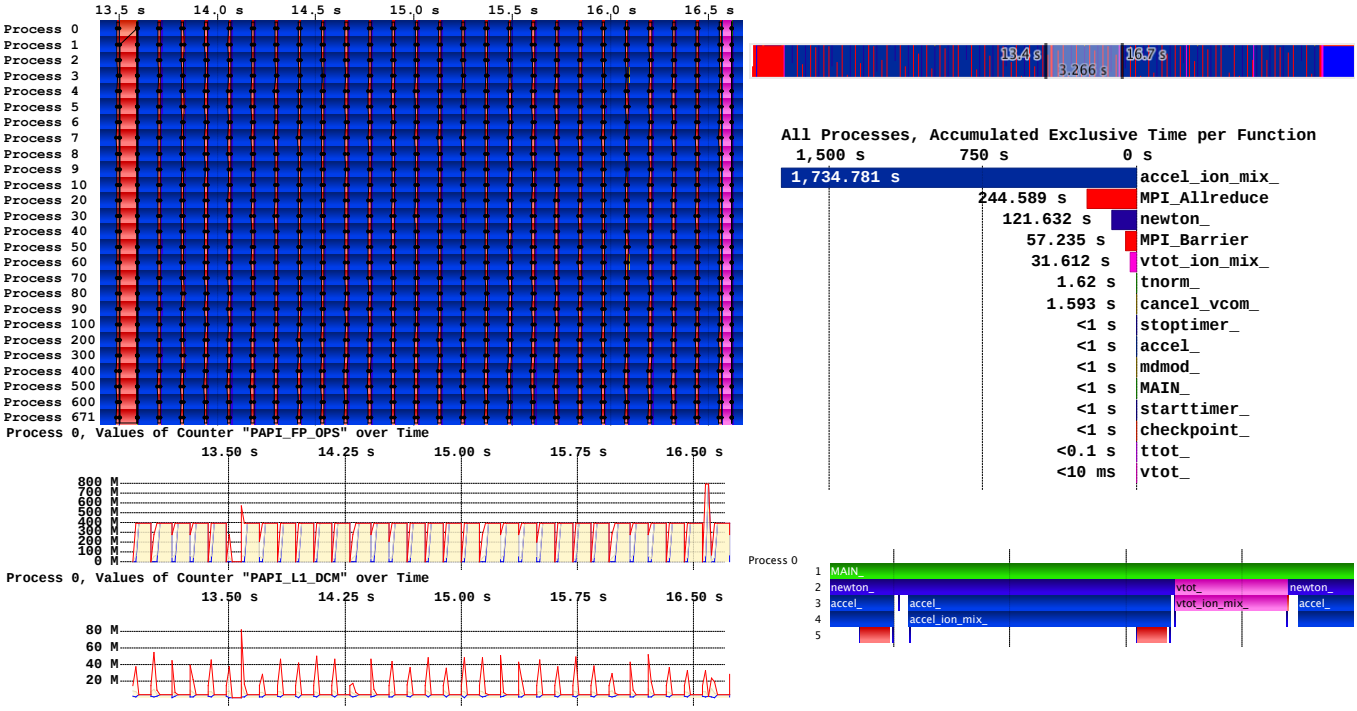

Figure 18. Vampir showing the MPI-only version running on all 672 cores.

the performance counter indicated a higher utilization of the hardware for other code blocks. First attempts to apply the Vampir framework showed heavy overhead induced by tracing OpenMP. We moved the OpenMP parallelization from the inner to the outer loop to increase workload per thread and to decrease OpenMP overhead. This improved the parallel efficiency and even the single threaded performance due to the change from 2 to 3 loops. A preliminary study on an dual socket, dual chip 16-threads AMD Interlagos system showed a 97.6% parallel efficiency up to 32 cores.

The changes to the source code will be included in future versions of the Spec OpenMP benchmark.

ACKNOWLEDGMENT

This document is based upon work supported in part by National Science Foundation (NSF) grant 0910812 to Indiana University for "FutureGrid: An Experimental, High-Performance Grid Test-bed." FutureGrid project partners include the University of California - San Diego and the San Diego Supercomputer Center (SDSC), the University of Chicago/Argonne National Labs, the University of Florida, Purdue University, the University of Southern California, the University of Texas - Austin, and the Center for Information Services and High Performance Computing at Technische Universität Dresden.

REFERENCES

[1] C. J. Horowitz, M. A. Pérez-García, J. Carriere, D. K. Berry, and J. Piekarewicz, "Nonuniform neutron-rich matter and coherent neutrino scattering," *Phys. Rev. C*, vol. 70, p. 065806, Dec 2004. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.70.065806

[2] C. J. Horowitz, J. Hughto, A. Schneider, and D. K. Berry, "Neutron Star Crust and Molecular Dynamics Simulation," 2011. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:1109.5095

[3] J. Hughto, A. S. Schneider, C. J. Horowitz, and D. K. Berry, "Diffusion of neon in white dwarf stars," *Phys. Rev. E*, vol. 82, p. 066401, Dec 2010. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.82.066401

[4] O. L. Caballero, S. Postnikov, C. J. Horowitz, and M. Prakash, "Shear viscosity of the outer crust of neutron stars: Ion contributions," *Phys. Rev. C*, vol. 78, p. 045805, Oct 2008. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.78.045805

[5] C. A. Stewart, M. R. Link, D. S. McCaulay, R. Henschel, and D. Y. Hancock, "Technical Report: Acceptance Test for FutureGrid Cray XT5m at Indiana University (Xray)," PTI, Technical Report PTI-TR12-005, Mar. 2012.

[6] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing Scalable Applications with Vampir, VampirServer and VampirTrace," in *Parallel Computing: Architectures, Algorithms and Applications*, ser. Advances in Parallel Computing, C. Bischof, M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, Eds., vol. 15. IOS Press, 2008, pp. 637–644. [Online]. Available: http://www.booksonline.iospress.nl/Content/View.aspx?piid=8455