Memory Scalability and Efficiency Analysis of Parallel Codes

Tomislav Janjusic and Christos Kartsaklis

Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831-6173
Email: {janjusict,kartsaklisc}@ornl.gov

***Abstract*—Memory scalability is an enduring problem and bottleneck that plagues many parallel codes. Parallel codes designed for High Performance Systems are typically designed over the span of several, and in some instances 10+, years. As a result, optimization practices which were appropriate for earlier systems may no longer be valid and thus require careful optimization consideration. Specifically, parallel codes whose memory footprint is a function of their scalability must be carefully considered for future exa-scale systems.**

In this paper we present a methodology and tool to study the memory scalability of parallel codes. Using our methodology we evaluate an application's memory footprint as a function of scalability, which we coined memory efficiency, and describe our results. In particular, using our in-house tools we can pinpoint the specific application components which contribute to the application's overall memory foot-print (application data-structures, libraries, etc.).

I. Introduction

Memory scalability is an enduring problem and bottleneck that plagues many parallel codes. Parallel codes designed for High Performance Systems are typically designed over the span of several, and in some instances over 10, years. As a result, optimization practices which were appropriate for earlier systems may no longer be valid and thus require careful optimization consideration. Specifically, parallel codes whose memory footprint is a function of their scalability must be carefully considered for future exa-scale systems.

The memory footprint is defined as the amount of memory that an application uses during its runtime. This includes the total amount of various application segments (code, data), as well as the allocation of dynamic and stack memories. We define memory efficiency as the metric describing the application's memory usage with respect to scalability. For example, an application whose memory footprint grows non-linearly may be described as having low memory efficiency. Ideally, an application's memory footprint will remain constant with respect to scalability. Memory efficiency is not to be confused with memory usage. An application's memory usage is a metric that describes how well an application is using the allocated memory. In this paper we do not focus on memory usage but rather memory allocation patterns as a function of scale.

In order to study the memory behavior of parallel codes we rely on several in-house tools. We developed and tweaked our tools specifically for this study and hope that their use will continue to be useful for other researchers too. Our tools rely on an instrumentation framework and a plug-in tool described here [1]. Using our tool we can pinpoint the specific application components which contribute to the application's overall memory footprint (eg. application data-structures, libraries, etc.) and trace their liveliness. For terminology accuracy we will define an object to be any allocated chunk of memory. An object group is a logical representation of two or more allocated objects that stem from the same module, sub-directory, or function. We stated earlier that our tools are based on binary instrumentation frameworks capable of tracing into applications' sub-structures and track their memory usage. This information is surprisingly underutilized as it does give great insight into the memory behavior of specific application phases and allows us to track the usage and footprint patterns. As a direct benefit we can observe if particular data-structures are being over-/ or under-utilized and if they are prone to increase in size at scale.

In this paper we present a methodology of a memory scalability study of parallel codes particularly focusing on dynamic memory behavior patterns, as well as memory overheads with respect to scalability. In order to make our case we have chosen an application that is known to have scalable performance, we proceed by evaluating its memory footprint as a function of scalability and by offering an attempt to quantify this behavior. We also elaborate on our methodology and our experimental setup.

Our study aims to expose the behavior of applications' memory allocation patterns as the applications scale. If appropriate, we can pinpoint specific application components which contribute to the inefficient use of memory thereby degrading

This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doepublicaccessplan).

the overall application performance. Our goal is to give a detailed profile of the application's components and by using our methodology we can potentially help steer critical future optimization. In this paper we do not make any claims about memory performance bottleneck manifestations.

The rest of the article is organized as follows: in Section II we will discuss related work and elaborate why scalability is a very important topic and we will present studies that complement this work. In Section III we will explain how our tools work and provide examples of collected data, and how to interpret the graphical representation of data. In Section IV we present a step by step analysis methodology for the application as well its accompanying message passing library. We conclude with tables showing regression models for the application and library as well as their sub-modules. In Section V we will summarize our findings and conclude the paper by offering future research directions as well as advise of potential memory bottlenecks likely to manifest as a result of scaling.

II. Related Work

Studying application performance encompasses a broad and extensively studied research area. It is well understood that application performance is the product of software and hardware that contribute to the application's overall performance. An application's execution time depends on the application's main code including any additional run-time system and library components or modules.

The research area of software performance is vast and ranges from addressing code-path optimizations, algorithm optimizations, to fine-grain memory layout optimizations. Performance and optimization research of HPC systems and HPC applications are undoubtedly addressing the memory aspects. This can range from memory usage, memory locality analysis, to memory footprints. Specifically, research in the area of memory scalability typically focuses on observing an application's overall memory consumption with respect to the growing number of processing elements. This research is generally facilitated using various tools such as Pin[2], Valgrind[3], Vampir Trace[4], PAPI hardware counters [5], or others. The majority of these tools as well the majority of related studies focus, however, on observing and explaining the overall memory consumption as a function of increased number of processing cores and seldom touch on identifying the root causes of a growing memory footprint. The detailed understanding of a software's memory footprint is thus left to the developers who are deeply engaged in the development cycle. The studies range from studying relatively small shared-memory SPEC OpenMP benchmarks [6], [7] to detailed studies of message passing interfaces such as MPICH2 [8], [9]. It is important to note that scalability analysis and memory consumption of widely used libraries such as OpenMPI has been extensively studied. The conclusions of such studies if convincing enough often nudge developers into redesigning their applications to facilitate the ever-increasing core count [10], [11].

Modern large-scale HPC system already consist of several hundred thousand cores, and it can be argued that as we approach the Exa-scale era the core count will likely be in the billions. Thus it is of vital importance to continue to study the effects of application's memory-scalability and develop models for future systems. Moreover, the current set of performance monitoring tools suffer from similar memory usage scalability problems, thus it is also increasingly important to address and engage with tool-developers to develop tools for Exa-scale system analysis. There are several efforts addressing the scalability of application performance monitoring tools such as WMTools [12], or by using built-in modules to help analyze codes such as MPI_T interface [13].

III. Analysis Framework

A. Memory Allocation Tracing

To complete this study we modified our in-house tools to generate the needed data, and developed a set of in-house post-processing tools for analysis purposes. Our main data collection tool is based on the Valgrind instrumentation framework [3] and a modified version of our memory-tracing tool, Gleipnir [14]. Gleipnir's unique ability to trace and map allocations to objects makes Gleipnir an ideal candidate tool. However, since we are not interested in generating the entire memory-transaction stream we modifed Gleipnir to make it faster and easier to use. We have previously studied the scalability of our tracing-tool in [1] and concluded that for the purposes of this study it meets our needs. The modified output of the tool is a trace of memory allocation and deallocation function calls mapped to an application's internal objects. We can choose to logically group our objects or create tree-like directory structures in order to identify root allocation modules.

```
X,1,MALLOC,018dcee70,196608,/lustre/atlas1/stf010/
proj-shared/janjust/OLCF/coral-benchmarks/LSMS_3_
rev237/include,Matrix_hpp_97,0

X,1,FREE,018dcee70,196608,Matrix_hpp_97,0
```

Fig. 1: An example malloc()/free() trace line.

Figure 1 shows a sample trace-line produced by our tool. The trace consists of intercepted allocation/deallocation function calls annotated with CSV meta-data. For example, in Figure 1 we can observe an intercepted *malloc()* call. The trace shows the *malloc()'s* returned pointer address *0x018dcee70*, the calls originating directory *../LSMS_3_rev237/include*, the originating file and line number *Matrix_hpp_97*, and finally the instance of the allocated object *0*. An object instance denotes the number of allocated objects, in this case this is the first instance of this object. Similarly, in the trace we will find deallocation functions, e.g. *free()*. The deallocation trace-line's meta-data consist of the pointer address of the to-be-freed object, the object name *Matrix_hpp_97*, and the objects instance. Note that tracking oject allocation/deallocation instances may be benefecial if we want to study potential fragmentation issues.

The tool generates a separate trace-file for every processing element (PE) or simply process. Thus we must post-process all trace-files in order to account for all memory related allocation/deallocation activities during an application's run-time. Because Valgrind is a binary instrumentation tool, similar to a virtual-machine, it does not provide run-time cycle accurate information. In order to get any time references we must use the number of instructions. Valgrind operates on a set of instructions known as super-blocks *SBs*. An SB consists of no more than 50 instructions. Thus, we can get an idea of timing by measuring the number of continuous superblocks. This, often overlooked information, is important because in order to understand an application's memory usage, and allocation/deallocation patterns we must get a sense of when they occur with respect to the total running time. Figure 2 shows a sample trace snippet of two malloc calls interleaved by approximately 120-200 instructions (4 SBs). Finally, the tool concludes every trace-file with a summary of invoked malloc family function calls, shown in Figure 3.

```
X,1,MALLOC,00d46ab80,24,/autofs/na4_sw/rhea/ \
openmpi/1.6.5/rhel6.4_gnu4.7.1/source/opal/  \
class,opal_object_c_122,0
X,1,SB_ENTRY
X,1,SB_ENTRY
X,1,SB_ENTRY
X,1,SB_ENTRY
X,1,MALLOC,00d46abc0,80,/autofs/na4_sw/rhea/ \
openmpi/1.6.5/rhel6.4_gnu4.7.1/source/opal/  \
class,opal_object_c_202,0
```

Fig. 2: An example malloc() with superblocks trace line.

B. Post-processing and Analysis

The set of post-processing tools consists of various scripts to analyze and aggregate the collected data into meaningful metrics. For example we can find the average number of allocation and deallocation calls for every object size, or the average life-time for every object size. We can choose to logically group or categorize individual objects based on our findings. We can also produce a high-level view of the application's memory usage data as we scale. This includes analyzing memory allocation patterns, peak memory usage, and object life-time analysis per process. We can then further categorize the memory usage based on application components e.g. the originating libraries.

The generated data and metrics allows us to reason about potential memory usage problems and provide a high-level view of the major memory usage contributors. However, using our trace meta-data we can drill into the root causes of memory usage and allocation patterns. Finally, the fine-grain breakdown of memory usage statistics comprises dissemination on a per object basis. The final step in our post-processing mechanism is to run a function-fit script to generate a model of object growth. The generated data is necessary in order to establish regression models on individual objects. We argue that this information gives application developers insights into the potential hazards that may arise as their software scales.

```
X,STATS
  total_lines:          157337
     flush_at: 18446744073709551615
total_flushes:               1

 malloc calls:           57881
 calloc calls:            2558
realloc calls:             672
   free calls:           50943

 Instructions:               0
        Loads:               0
       Stores:               0
     Modifies:               0
--
```

Fig. 3: A malloc et al. summary for every trace-file.

C. Memory Allocation and Usage Patterns

Dynamic memory allocation and memory management is a ubiquitous process in virtually all computing systems. Due to the strain on the memory system high-performance applications are especially sensitive to memory mismanagement and over-allocation of memory objects. A good memory allocation scheme, and by implication good memory allocation management, must carefully consider the properties of standard memory allocators as well as the impact of memory allocation on the overall system performance. The nature of dynamic memory requests make allocation algorithms complex and it has been shown that for any allocation algorithm there exists a worst-case allocation pattern [15].

Long running application such as HPC applications are likely to be severely affected by allocation patterns. The study in [15] offers a review and critique of well known allocation algorithms. It also shows some well known allocation patterns. As a general rule most application will experience three distinct allocation patterns:

1) *Peak* memory allocation is observed when memory is continuously allocated and freed in short bursts. If the application shuffles objects of varying sizes this behavior can lead to high external fragmentation.
2) *Plateau* memory allocation is observed when applications allocate constant blocks of memory and frees the memory at the end. We expect that most high-performance applications will observe this type of allocation pattern.
3) *Ramp* memory allocation pattern is observed in applications that continuously allocate memory resulting in a steady increase in memory usage. If the memory requests exceed available system resources the application will either run out of space or cause heavy memory swaps.

Figure 4 illustrates what a potential application's allocation pattern may look like. Note that the figure only shows a high-level view of a single process split into memory usage categories of: Application, Communication library (OpenMPI), and Other−reserved for allocated but unknown fragments of memory. The x-axis shows the number of allocations, and the y-axis shows the number of currently allocated bytes. In this example we can observe that during the majority of allocation/deallocation events the application maintained a constant memory consumption[1]. In the final stages of the allocation pattern we notice a burst of memory usage in both the application and the MPI library. The burst in memory usage could be an indicator of a distinct phase in the application's execution.

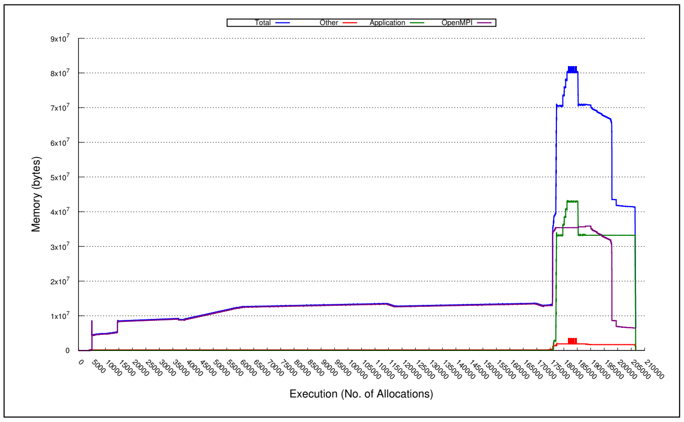

Fig. 4: Memory allocation pattern example.

In order to understand the memory usage, scalability, and efficiency in terms of scalability we ask the following questions:

1) Which objects are the primary drivers of memory usage?
2) Do all processes exhibit similar allocation patterns?
3) Does peak or average memory usage change with respect to scalability?
4) If 3) is true, can we reason about which objects are the primary drivers of the overall memory usage as a function of scalability and how fast do they grow?

As a final note, we stress that memory allocation effects are intensified in parallel applications because of the added complexity. Therefore any application whose memory behavior is a function of scale most carefully analyze its memory usage in order to avoid potential bottlenecks..

IV. Scalability Analysis

A. Experimental setup

We conducted our memory scalability analysis using ORNL's Rhea system. Rhea is a (512)-node commodity-type Linux cluster. Each of Rhea's nodes contain two 8-core 2.0 GHz Intel Xeon processors with Hyper-Threading and 64GB of main memory. Rhea is connected to the OLCF's 32PB high performance Lustre filesystem "Atlas". For purposes of this study we chose the LSMS [16] benchmark from the set of CORAL-benchmarks. Because of the benchmarks ease of deployment and scalability LSMS is a code that characterize both single node performance and full system scalability.

[1]A more careful analysis of Figure 4 shows that during the execution several smaller blocks are allocated and deallocated in short bursts; however, due to the high-level view this behavior is hidden to the naked eye.

B. Total Peak Memory Usage

We start by observing the application peak memory performance as we scale the application from 1 to 128 nodes. Figure 5 shows the application's total memory usage as well as the major memory contributors. We categorized the main memory usage components based on their origin. In this case we have only two: the application and the OpenMPI library. We can observe from Figure 5 that the overall memory consumption is decreasing. This is to be expected since we are employing strong scaling. That is, we do not increase the problem size with the number of nodes. Notice that while overall memory consumption is decreasing, the communication library's memory footprint is increasing. This too is expected because of the added complexity when communicating with an increased number of processing elements. Our question is thus: are all application's objects decreasing as a function of scale, and which objects of the OpenMPI library are increasing and how quickly?

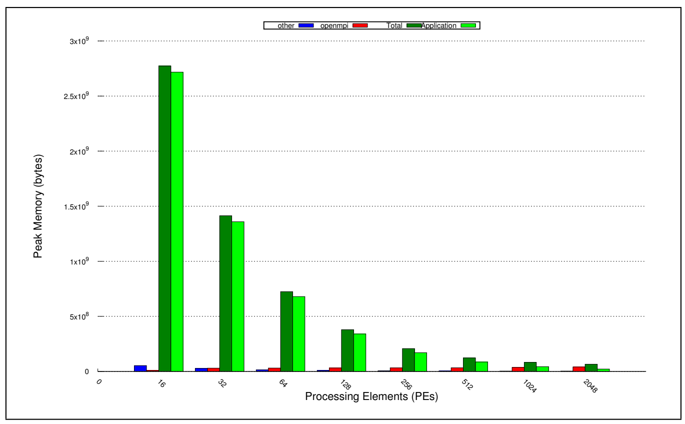

Fig. 5: Total peak memory usage.

C. Application Peak Memory

We will start by analyzing the application's major memory footprint contributors. For this purpose we can categorize the memory allocation origin and plot the results. The LSMS' benchmark directory structure is shown in Table I. We will use the directory structure as basis in order to determine the main drivers of memory footprint as well observe any changes in allocation frequency and size as a function of scale. We found that the major memory contributors originates from the */include* directory.

Figure 6 show the peak memory usage of all objects originating from the */include* sub directory. We notice that the memory footprint reduces significantly as we increase the number of processing elements almost identical to what we observe in Figure 5. However, even with memory decay over the number of processing elements, can we determine if any objects are in fact growing due to the added complexity of more processing elements?

```
LSMS / src / Core
     |       Potential
     |       Communication
     |       TotalEnergy
     |       SingleSite
     |       Misc
     |       etc.
     / include
     / lua
```

TABLE I: LSMS directory structure

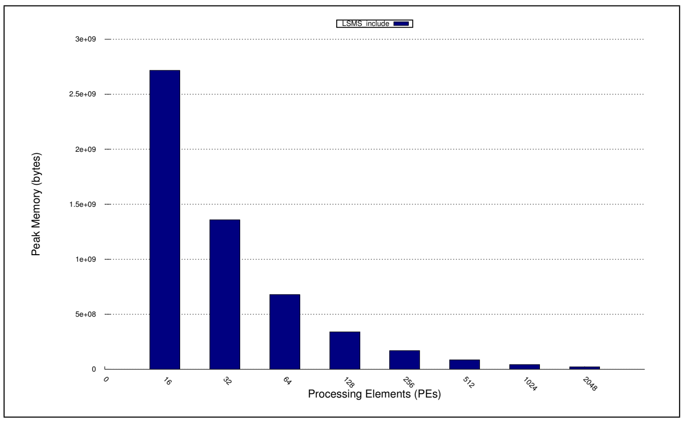

Fig. 6: Peak memory usage (LSMS/include).

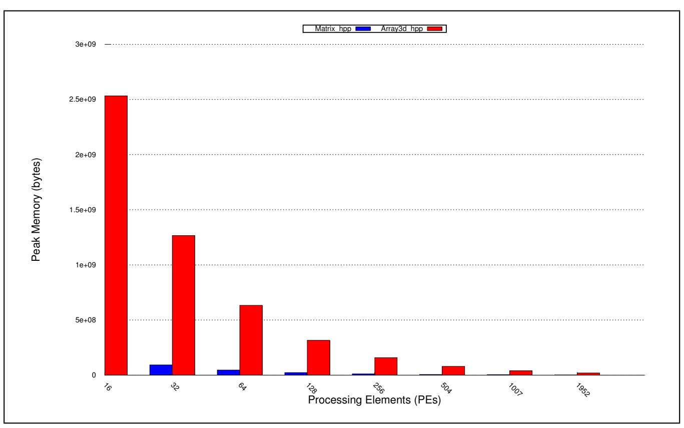

Fig. 7: Peak memory usage (main objects).

Figure 7 shows objects from the */include* directory further categorized based on their file of origin. These objects are named "Matrix_hpp", and "Array3d_hpp". Note that these represent the files of the originating blocks, in theory we can drill even further to determine exactly which objects they are based on the originating function and source-code line number; however, as we will show later, in this example this is not necessary in order to understand the application's memory footprint.

For most practical purposes we can already observe the application's scalability memory footprint behavior. However we must also consider smaller objects whose memory footprint behavior may go unnoticed. In this example these objects originate from various sources of the directory structure. We have isolated these objects into: */lua, /src/*$\{$*Core, Potential, Communication, TotalEnergy, etc.* $\}$ sub-directories. The peak memory usage of these objects is shown in Figure 8. We can observe a nearly constant memory consumption for all remaining objects rendering this application highly scalable. In fact, to our knowledge the only memory bottleneck that may cause this application to under-perform is the amount of available memory per node.

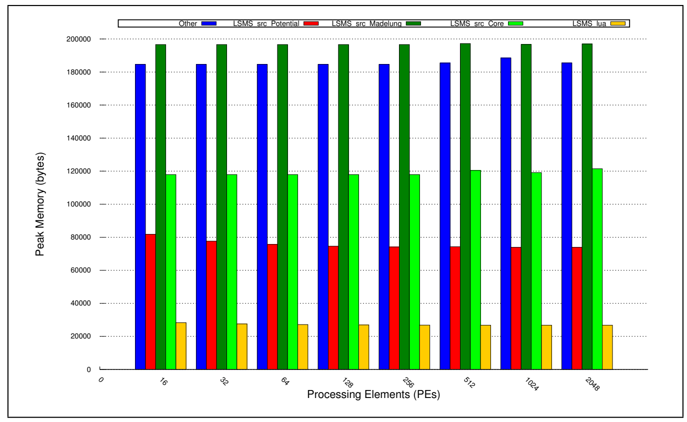

Fig. 8: Peak memory usage (smaller memory allocations).

Object	Regression, f(x) =	r^2	r
src_VORPOL	$960298.79 + 3.71 \times x$	0.372	0.610
src_Core	$117964.54 + 1.71 \times x$	0.771	0.878
Other	$184820.81 + 0.97 \times x$	0.259	0.509
src_MultipleScattering	$74812.35 + 0.18 \times x$	0.855	0.925
src_Madelung	$196656.03 + 0.17 \times x$	0.248	0.498
src_TotalEnergy	$67296.12 + 0.05 \times x$	0.470	0.685
src_Communication	$65532.75 + 0.01 \times x$	0.770	0.877
src_Main	$8214.09 + -0.00 \times x$	0.063	0.251
src_Potential	$76719.11 + -1.91 \times x$	0.243	0.493
lua	$27337.28 + -0.40 \times x$	0.268	0.518
include	$1.77 \times 10^9 + -687024.04 \times x$	0.268	0.518

TABLE II: Application sub-directories regression table.

Table II shows the regression fit table for the major sub-directory components. The fastest growing component is the *src/VORPOL* module, followed by objects in *src/Core*. Notice that observations in Figure 8 show that most smaller objects remain fairly constant; however, the regression table shows us that they are growing in fact.

In Figure 9 we can observe the memory allocation pattern for a single process during the application's execution. The y-axis shows the current amount of allocated memory in bytes, the x-axis shows the number of allocations. We can clearly observe the allocation pattern for the larger object. However, we can also observe a large number of allocations and deallocations for smaller objects. Generally this is an undesired behavior, because a large number of allocations and deallocations can lead to external memory fragmentation. Moreover, depending on the allocator's performance a large number of reallocations can severely degrade performance, especially for long running applications. While we only show a single process allocation and – minding that for obvious reasons it would be impractical to show for every process from the running process pool – we also noticed that for most processes the behavior shown in Figure 9 is uniform. That is to say, virtually all processing elements observe the same allocation pattern.

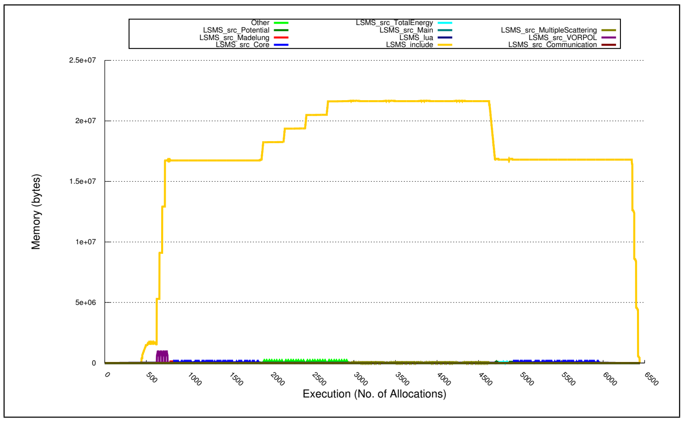

Fig. 9: Memory usage during process execution (main objects, single process).

In Figure 10 we show the memory allocation pattern for smaller objects by omitting allocated objects from */include*. We can clearly observe the allocation frequency. The allocation pattern shown in Figure 10 is an example of *peak* memory allocation pattern.

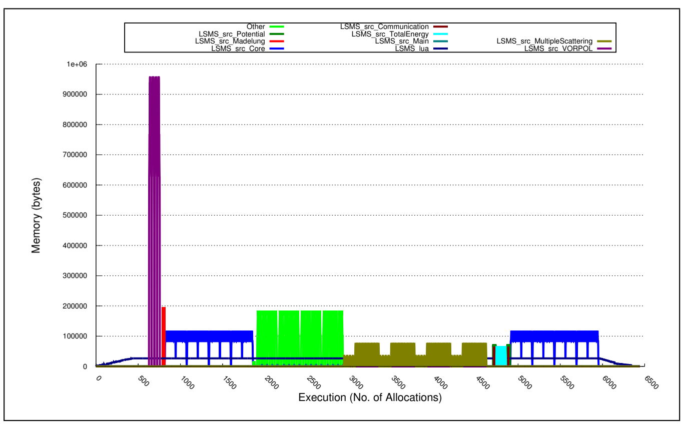

Fig. 10: Memory usage during process execution (without main objects, single process).

The general rule on allocation patterns is that smaller objects are allocated and deallocated more frequently than larger objects [15]. This can also be an indicator of an object's life-time. In order to test this, we formulated our plots to show object's average life-time vs. their size. In Figure 11 we show the average lifetime in superblocks versus the objects size. It is somewhat puzzling that larger objects, presumably objects that comprise the */include* category are experiencing a relatively short average life-time. Similarly, in Figure 12 we show the number of allocation/deallocations for specific object sizes. We can observe that smaller objects have a significantly smaller number of allocation and deallocations occurring. This behavior is consistent with Figure 10. We are still unsure as to why we are experiencing a short life-time behavior on larger objects and continue to investigate.

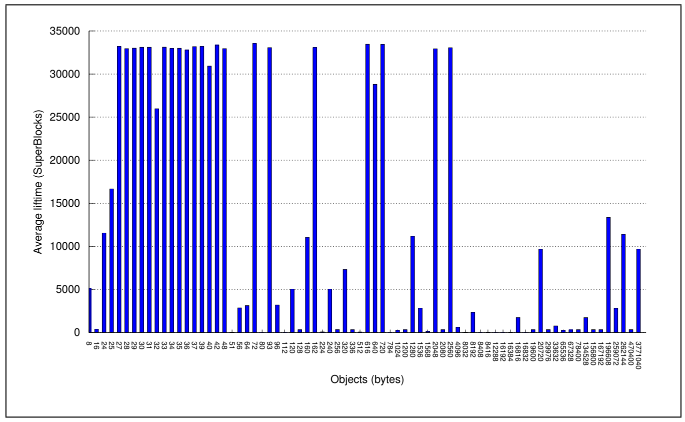

Fig. 11: Object's average life-time.

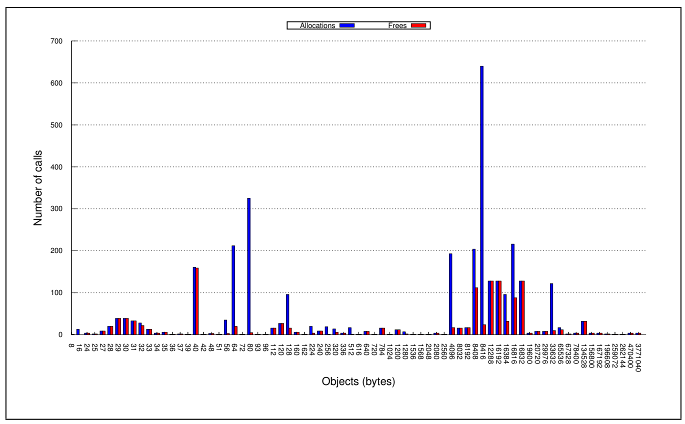

Fig. 12: Object's number of allocations/deallocations.

D. OpenMPI Peak Memory

Unlike the memory behavior of the main application, the OpenMPI library's memory footprint is a function of scale. That means, that due to the control structures which must be allocated in order to allow communication of various processing elements the memory footprint of the library is increasing. This behavior is visible in the overall memory footprint figure, Figure 5. Similarly to the previous subsection we can decompose the components of the overall memory consumption by sub-directory structure. In OpenMPI we find three distinct memory consumers. The Open run-time system (ORTE), Open Portable Access Layer (OPAL), and OpenMPI (OPMI). Figure 13 shows the memory footprint decomposition based on those three modules.

In Figure 13 we can observe that all three modules experience an increase in peak memory usage with an increased number of processes. The largest and fastest growing contributor is the OMPI module. OPAL remains constant except when the number of nodes doubles from 1024 PEs (64 nodes) to 2048 PEs (128 nodes). Albeit smaller, ORTE's memory consumption also increases.

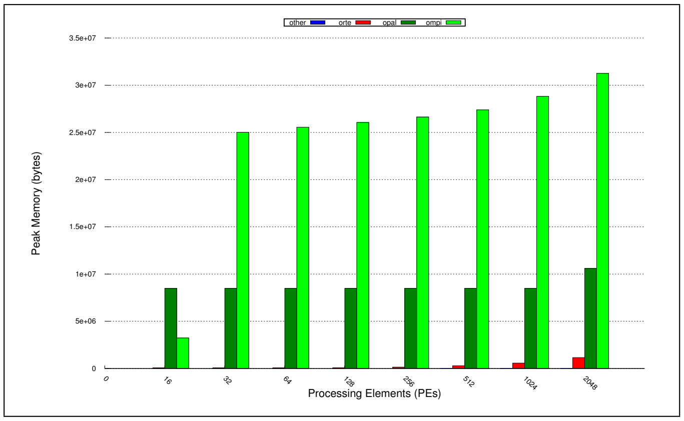

Fig. 13: Peak memory usage (openmpi).

object	regression	r^2	r
ompi	$y = 25560598.851 + 2906.177 \times x$	0.971	0.985
opal	$y = 8237864.787 + 952.509 \times x$	0.777	0.882
orte	$y = 14114.224 + 552.774 \times x$	0.999	1.000
other	$y = -4.259 + 0.504 \times x$	0.912	0.955

TABLE III: OpenMPI subdirectories regression table.

Similarly to the previous sub-section we can show OpenMPI's memory allocation pattern as a function of time. Figure 14 shows the memory allocation pattern for the OpenMPI library for a single process. We can observe a spike in memory consumption by the OMPI module. This can be an indication of a communication phase in the application. Recall that we previously stated that allocation patterns are uniform across processing elements. While that was true for the main application's objects, OpenMPI has different behavior. We noticed that certain individual tracefiles are different in size (See tracefile structure in Figure 4. This can mean one of two things: 1) The process runs longer, meaning that more superblocks are processed, or 2) There are more allocation and deallocation calls taking place.

Table III shows the regression formulas derived from our data. It shows that all modules have relatively strong growth and thus are likely to exert heavy memory usage when scaled to Exa-scale. As we explained earlier we can further distill into individual modules to find the running culprits of memory usage as well as memory growth.

Our analysis shows that when the application started executing on multiple nodes we found one processes whose tracefile is larger relative to other processes, meaning that the process is performing additional allocation and deallocation calls. From previous figures we can observe that OpenMPI's peak-memory consumption increases significantly when executing on multiple nodes (Figure 5). This behavior is expected because of the added computational requirements to coordinate messages across multiple nodes. The reason of why this is happening is outside the scope of this paper, however, for reference we included Figure 15 that shows the seemingly abnormal behavior. Specifically the Figure shows that this process' OpenMPI memory consumption is growing linearly. There can be various reasons behind this behavior; however, we must stress that this peak memory abnormality does not negatively impact overall peak memory usage. This is because the scripts that aggregate our data will discard or average any peak-memory outliers.

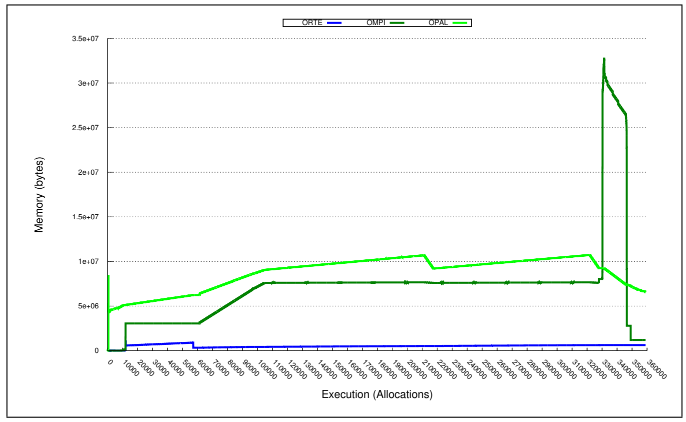

Fig. 14: Memory usage during process execution (OMPI, ORTE, OPAL, single process).

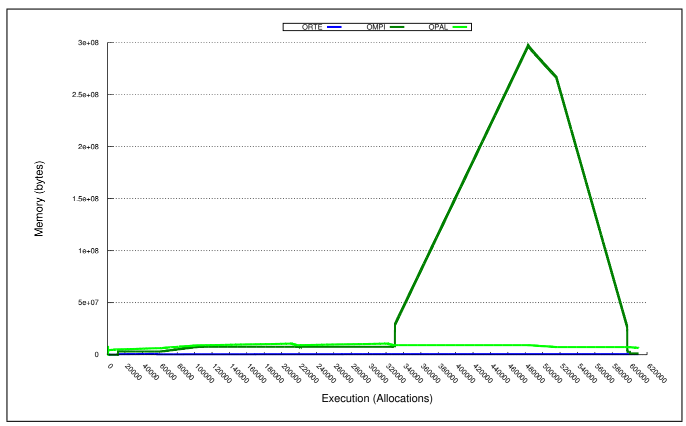

Fig. 15: Memory usage during process execution (OMPI, ORTE, OPAL, single process).

Table IV shows the derived regression formulas of object groups that stem from the OMPI module. An object group is a logical grouping of individual object allocation per originating file or function. We can observe that the largest contributor to object growth is the *btl_openib_endpoint_c* file, followed by object groups *bml_base_btl_c, pml_ob1_comm_c, etc*. The reasons behind the object growth is beyond this paper; however, we hope that our methodology for memory efficiency as a function of scale analysis may yield an interesting way of studying software memory footprints.

Object	Regression, f(x) =	r^2	r
btl_openib_endpoint_c	$-16945.79 + 1743.03 \times x$	1.000	1.000
bml_base_btl_c	$-3443.84 + 377.17 \times x$	1.000	1.000
pml_ob1_comm_c	$-1978.89 + 242.80 \times x$	1.000	1.000
ompi_free_list_c	$2369880.18 + 241.47 \times x$	0.677	0.823
mpool_rdma_module_c	$22725836.32 + 178.43 \times x$	0.284	0.533
btl_openib_proc_c	$-1416.56 + 140.39 \times x$	1.000	1.000
btl_openib_connect_oob_c	$61311.70 + 42.20 \times x$	0.714	0.845
btl_tcp_proc_c	$730.29 + 25.14 \times x$	1.000	1.000
allocator_bucket_alloc_c	$241836.98 + 22.60 \times x$	0.185	0.430
coll_tuned_module_c	$414.86 + 16.76 \times x$	1.000	1.000
coll_basic_module_c	$-137.14 + 16.76 \times x$	1.000	1.000
bml_r2_c	$-73.14 + 16.76 \times x$	1.000	1.000
btl_openib_component_c	$28189.19 + 4.07 \times x$	0.632	0.795
coll_tuned_allreduce_c	$1863.49 + -1.25 \times x$	0.354	0.595
btl_openib_connect_rdmacm_c	$906.20 + -0.00 \times x$	0.037	0.193
rcache_vma_component_c	$664.00 + 0.00 \times x$	1.000	1.000
mpool_rdma_component_c	$656.00 + 0.00 \times x$	1.000	1.000
dpm_base_common_fns_c	$117.00 + 0.00 \times x$	1.000	1.000
coll_tuned_bcast_c	$11.21 + 0.00 \times x$	0.035	0.187
btl_sm_component_c	$16.00 + 0.00 \times x$	1.000	1.000
btl_openib_mca_c	$245.00 + 0.00 \times x$	1.000	1.000
btl_openib_lex_c	$82002.00 + 0.00 \times x$	1.000	1.000
btl_openib_fd_c	$368.00 + 0.00 \times x$	1.000	1.000
allocator_bucket_c	$319.79 + -0.00 \times x$	0.035	0.187
btl_openib_ip_c	$9772.00 + 0.00 \times x$	1.000	1.000
btl_openib_async_c	$32.00 + 0.00 \times x$	1.000	1.000
mpool_sm_component_c	$322.00 + 0.00 \times x$	0.192	0.438
btl_openib_connect_base_c	$279.00 + 0.00 \times x$	1.000	1.000
proc_c	$-256.85 + 25.32 \times x$	1.000	1.000
pml_base_select_c	$48.00 + 0.00 \times x$	1.000	1.000
btl_self_component_c	$8.00 + 0.00 \times x$	1.000	1.000
btl_tcp_component_c	$912.00 + 0.00 \times x$	1.000	1.000
btl_sm_c	$2435.79 + -0.00 \times x$	0.035	0.187
pml_base_open_c	$7.00 + 0.00 \times x$	1.000	1.000
btl_openib_ini_c	$5627.00 + 0.00 \times x$	1.000	1.000
btl_base_mca_c	$280.00 + 0.00 \times x$	1.000	1.000
pml_v_output_c	$123.07 + 0.00 \times x$	0.208	0.456
coll_tuned_topo_c	$444.40 + -0.00 \times x$	0.912	0.955
btl_openib_c	$50880.12 + -0.00 \times x$	0.764	0.874

TABLE IV: OpenMPI OMPI regression table.

Object	Regression, f(x) =	r^2	r
oob_tcp_msg_c	$14160.46 + 573.46 \times x$	0.997	0.999
grpcomm_base_modex_c	$-2463.98 + 317.05 \times x$	1.000	1.000
nidmap_c	$-62.55 + 8.37 \times x$	1.000	1.000
session_dir_c	$1272.25 + 0.00 \times x$	0.511	0.715
orte_dt_packing_fns_c	$4.00 + 0.00 \times x$	1.000	1.000
oob_tcp_c	$256.00 + 0.00 \times x$	1.000	1.000
name_fns_c	$124.31 + 0.00 \times x$	0.714	0.845
proc_info_c	$135.94 + 0.00 \times x$	0.235	0.485
rml_oob_contact_c	$264.83 + 0.00 \times x$	0.419	0.648
rml_base_contact_c	$64.46 + 0.00 \times x$	0.472	0.687
orte_dt_unpacking_fns_c	$8.00 + 0.00 \times x$	1.000	1.000
rml_oob_send_c	$32.00 + 0.00 \times x$	1.000	1.000
rml_oob_recv_c	$96.00 + 0.00 \times x$	1.000	1.000
rml_oob_component_c	$128.00 + 0.00 \times x$	1.000	1.000
oob_tcp_addr_c	$256.00 + 0.00 \times x$	1.000	1.000
orte_mca_params_c	$169.00 + 0.00 \times x$	1.000	1.000

TABLE V: OpenMPI ORTE regression table.

Table V shows the ORTE module's object growth. Here too we can observe that in fact some objects have peak memory usage as function of scale. Similarly, we have also analyzed and summarized regression functions for objects that originated from the OPAL module, shown in Table VI.

V. Conclusions

Understanding application's memory scalability is important for any application that targets Exa-scale. Even current application's which are developed for current system may benefit from understanding the memory efficiency using methodologies present in this paper. Because most application's run on multiple systems with different memory footprint and memory hierarchies. This makes our methodology and tools developed for this study of special interest for HPC applications and developers who must have efficient memory scalability built in, or planning on targeting future systems.

As part of this study we developed a modified version of our tracing tool as well as engaged with industry partners to develop a more robust mechanism to run codes at an even greater number of cores. Similarly, this study forms a solid base for more in-depth research required to form more rigorous memory scalability modeling. The goal of this paper is to present a methodology to study memory efficiency as a function of scalability and to present a mechanism (or tool) to trace memory allocation patterns.

We believe that such information is of vital interest to the broader scientific community and welcome future collabo-

Object	Regression, f(x) =	r^2	r
opal_object_h	$18879.44 + 2792.00 \times x$	1.000	1.000
dss_unpack_c	$19499.00 + 44.04 \times x$	0.944	0.972
opal_pointer_array_c	$9662.07 + 30.44 \times x$	1.000	1.000
dss_internal_functions_c	$38027.87 + 10.80 \times x$	0.981	0.991
argv_c	$1376.61 + 2.37 \times x$	0.823	0.907
output_c	$307.33 + 0.01 \times x$	0.456	0.676
opal_hash_table_c	$127776.39 + -0.04 \times x$	0.757	0.870
opal_datatype_optimize_c	$2208.00 + 0.00 \times x$	1.000	1.000
timer_linux_component_c	$568.00 + 0.00 \times x$	1.000	1.000
topology_c	$11520.86 + 0.00 \times x$	0.365	0.604
topology-xml_c	$26412.86 + 0.00 \times x$	0.365	0.604
slist_c	$32.00 + 0.00 \times x$	1.000	1.000
opal_convertor_c	$664.00 + 0.00 \times x$	1.000	1.000
opal_progress_c	$32.00 + 0.00 \times x$	1.000	1.000
opal_params_c	$115.00 + 0.00 \times x$	1.000	1.000
opal_value_array_h	$152.00 + 0.00 \times x$	1.000	1.000
signal_c	$152.00 + 0.00 \times x$	1.000	1.000
lt__alloc_c	$438.00 + 0.00 \times x$	1.000	1.000
if_c	$800.00 + 0.00 \times x$	1.000	1.000
epoll_c	$928.00 + 0.00 \times x$	1.000	1.000
cpuset_c	$54296.00 + 0.00 \times x$	1.000	1.000
event_c	$1600.00 + 0.00 \times x$	1.000	1.000
dss_register_c	$471.00 + 0.00 \times x$	1.000	1.000
distances_c	$0.00 + 0.00 \times x$	1.000	1.000
private_h	$25056.00 + 0.00 \times x$	1.000	1.000
path_c	$4.00 + 0.00 \times x$	1.000	1.000
os_dirpath_c	$32816.00 + 0.00 \times x$	1.000	1.000
net_c	$40.00 + 0.00 \times x$	1.000	1.000
mca_base_param_c	$96613.14 + -0.00 \times x$	0.191	0.437
mca_base_components_open_c	$2756.90 + 0.00 \times x$	0.757	0.870
opal_free_list_c	$30039.41 + -0.00 \times x$	0.122	0.350
opal_datatype_create_c	$2016.00 + 0.00 \times x$	1.000	1.000
keyval_lex_c	$82002.00 + 0.00 \times x$	1.000	1.000
os_path_c	$193.98 + 0.00 \times x$	0.546	0.739
basename_c	$96.21 + 0.00 \times x$	0.556	0.746
opal_value_array_c	$138992.00 + 0.00 \times x$	1.000	1.000
opal_bitmap_c	$3.02 + 0.13 \times x$	1.000	1.000
memory_linux_ptmalloc2_c	$8388640.00 + 0.00 \times x$	1.000	1.000
keyval_parse_c	$568.00 + 0.00 \times x$	1.000	1.000
mca_base_open_c	$306.05 + 0.00 \times x$	0.228	0.478
carto_base_graph_c	$7.00 + 0.00 \times x$	1.000	1.000
topology-linux_c	$32856.00 + 0.00 \times x$	1.000	1.000
shmem_mmap_module_c	$4.00 + 0.00 \times x$	1.000	1.000
installdirs_base_expand_c	$1318.00 + 0.00 \times x$	1.000	1.000
opal_object_c	$9519.62 + -0.00 \times x$	0.540	0.735
mca_base_component_compare_c	$115.00 + 0.00 \times x$	1.000	1.000
ltdl_c	$32937.00 + 0.00 \times x$	1.000	1.000

TABLE VI: OpenMPI OPAL regression table.

rations on specific applications to study memory efficiency. Finally, a very preliminary and worrisome projection of the memory footprint using the regression formulas, but ignoring topology and density, for the current two largest systems is shown in Table VII.

Objects	Titan 300k cores	Tianhe-2 384k cores	Hypothetical 100m cores
src_VORPOL	20 MB	23 MB	372 MB
ompi	897 MB	1.14 GB	290 GB
opal	293 MB	374 MB	95 GB
orte	166 MB	212 MB	69 GB

TABLE VII: Per core memory footprint projections using regression tables in the context of LSMS.

REFERENCES

[1] D. W. Tomislav Janjusic, Christos Kartsaklis, "Scalability analaysis of gleipnir, a memory tracing tool, on titan," in *Cray User Group (CUG)*, Lugano, Switzerland, May 2014.

[2] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Janapa, and R. K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *In Programming Language Design and Implementation*. ACM Press, 2005, pp. 190–200.

[3] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *SIGPLAN Not.*, vol. 42, pp. 89–100, June 2007. [Online]. Available: http://doi.acm.org/10.1145/1273442.1250746

[4] M. S. Muller, A. Knupfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing scalable applications with vampir, vampirserver and vampirtrace." in *PARCO*, ser. Advances in Parallel Computing, vol. 15. IOS Press, 2007, pp. 637–644.

[5] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.

[6] D. Molka, R. Schöne, D. Hackenberg, and M. S. Müller, "Memory performance and spec openmp scalability on quad-socket x86 64 systems," in *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ser. ICA3PP'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 170–181.

[7] K. Fürlinger, M. Gerndt, and J. Dongarra, "Scalability analysis of the spec openmp benchmarks on large-scale shared memory multiprocessors," in *Proceedings of the 7th International Conference on Computational Science, Part II*, ser. ICCS '07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 815–822.

[8] D. Goodell, W. Gropp, X. Zhao, and R. Thakur, "Scalable memory use in mpi: A case study with mpich2," in *Proceedings of the 18th European MPI Users' Group Conference on Recent Advances in the Message Passing Interface*, ser. EuroMPI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 140–149.

[9] O. Perks, D. Beckingsale, A. Dawes, J. Herdman, C. Mazauric, and S. Jarvis, "Analysing the influence of infiniband choice on openmpi memory consumption," in *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, July 2013, pp. 186–193.

[10] M. Luo, H. Wang, J. Vienne, and D. K. Panda, "Redesigning mpi shared memory communication for large multi-core architecture," *Comput. Sci.*, vol. 28, no. 2-3, pp. 137–146, May 2013. [Online]. Available: http://dx.doi.org/10.1007/s00450-012-0210-8

[11] P. Shamis, R. Graham, M. Venkata, and J. Ladd, "Design and implementation of broadcast algorithms for extreme-scale systems," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on*, Sept 2011, pp. 74–83.

[12] O. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis, "Wmtools - assessing parallel application memory utilisation at scale," in *Proceedings of the 8th European Conference on Computer Performance Engineering*, ser. EPEW'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 148–162.

[13] R. Rajachandrasekar, J. Perkins, K. Hamidouche, M. Arnold, and D. K. Panda, "Understanding the memory-utilization of mpi libraries: Challenges and designs in implementing the mpi_t interface," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14. New York, NY, USA: ACM, 2014, pp. 97:97–97:102.

[14] T. Janjusic, K. M. Kavi, and B. Potter, "International conference on computational science, iccs 2011 gleipnir: A memory analysis tool," *Procedia CS*, vol. 4, pp. 2058–2067, 2011.

[15] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review." Springer-Verlag, 1995, pp. 1–116.

[16] "Future proofing wl-lsms: Preparing for first principles thermodynamics calculations on accelerator and multicore architectures," in *Cray User Group (CUG)*, Fairbanks, Alaska, May 2011.