APPLICATION-LEVEL POWER MONITORING AND MANAGEMENT IN HIGH PERFORMANCE COMPUTING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
IVANA MARINCIC

CHICAGO, ILLINOIS
MAY 2018
# TABLE OF CONTENTS

LIST OF FIGURES ................................................................. iii
LIST OF TABLES ................................................................. v
ACKNOWLEDGMENTS .............................................................. vi
ABSTRACT ................................................................................ vii

1 INTRODUCTION AND MOTIVATION ...................................... 1

2 POLIMER: AN ENERGY MONITORING AND POWER CONTROL API .... 4
  2.1 Power Monitoring and Limiting in HPC: Background and Related Work .... 6
  2.2 PoLiMEr: General Framework .................................................. 8
  2.3 Power Monitoring and Limiting on the XC40 .................................... 9
  2.4 PoLiMEr Usage ................................................................. 10
  2.5 PoLiMEr Output ............................................................... 13

3 POLIMER EVALUATION ....................................................... 15
  3.1 Characterizing Application Power Consumption Behavior using PoLiMEr ... 15
  3.2 Power Capping with PoLiMEr .................................................. 18
  3.3 Power Capping with PoLiMEr: An Architectural Perspective ................. 20
  3.4 Overhead Measurements ..................................................... 21

4 POWER ALLOCATION IN IN-SITU ANALYSIS ......................... 23
  4.1 Related Work .................................................................... 23
  4.2 The Structure of In-situ Analysis and Impacts of Power Constraints ............ 25
  4.3 Power Allocation Between Simulation and Analysis .............................. 26

5 EVALUATION OF IN-SITU POWER ALLOCATION ..................... 31
  5.1 Integrating PoLiMEr with LAMMPS .......................................... 31
  5.2 Evaluation Setup and Results ................................................... 33
    5.2.1 Initial power distribution: analysis has more power .................... 34
    5.2.2 Initial power distribution: simulation has more power ................ 38
    5.2.3 Initial power distribution: simulation and analysis have equal power .... 45

6 FUTURE WORK AND CONCLUSION ....................................... 47
  6.1 Conclusion ........................................................................ 49
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Idle power consumption on a single node on Theta over time across power domains measured by PoLiMEr.</td>
<td>14</td>
</tr>
<tr>
<td>3.1</td>
<td>LAMMPS Power Trace. 128 nodes, 64 ranks/node, 2 threads/rank. Rhodopshin benchmark with $6 \times 6 \times 6$ repetitions (6.9 million atoms total) and 100 iterations.</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Average energy consumption per tag per iteration.</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>Average time per tag per iteration.</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Energy and time values per tag per iteration for LAMMPS on 128 nodes.</td>
<td>17</td>
</tr>
<tr>
<td>3.5</td>
<td>128 Nodes</td>
<td>18</td>
</tr>
<tr>
<td>3.6</td>
<td>1024 Nodes</td>
<td>18</td>
</tr>
<tr>
<td>3.7</td>
<td>LAMMPS total energy consumption and time per node for two runs using 128 and 1024 nodes respectively.</td>
<td>18</td>
</tr>
<tr>
<td>3.8</td>
<td>Intel MKL DGEMM executed on a single node, 1000 iterations ($1024 \times 1024$ matrix size). RAPL package power cap was increased every 100 iterations by 20 W, starting with 100 W. Vertical lines mark increase in power cap.</td>
<td>19</td>
</tr>
<tr>
<td>3.9</td>
<td>No power cap</td>
<td>19</td>
</tr>
<tr>
<td>3.10</td>
<td>110 W RAPL package power cap</td>
<td>19</td>
</tr>
<tr>
<td>3.11</td>
<td>Power consumption trace of STREAM using DDR memory on 64 nodes (1 MPI rank/node, 64 threads/rank, 1 thread/core), without and with power cap. Flat memory mode, quad cluster mode on KNL. 15 GB of memory, 100 iterations.</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Diagram of power consumption over time of a particular compute step between two synchronization points. The red line indicates the global power budget. On the left, analysis finishes sooner and waits for the simulation to reach the synchronization point. On the right, power is reallocated from analysis to speed up simulation such that both can synchronize sooner than with the initial power distribution.</td>
<td>26</td>
</tr>
<tr>
<td>5.1</td>
<td>Runtime of LAMMPS with PoLiMEr’s power allocation against baseline. Analysis initially assigned more power than simulation.</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Trace of power consumption of analysis and simulation running on 32 nodes each. PoLiMEr reallocates power every 40 simulation steps, gradually increasing performance compared to the baseline that maintains initial power allocation. Time and power axis are at the same scale.</td>
<td>36</td>
</tr>
<tr>
<td>5.3</td>
<td>Difference between analysis and simulation in time to reach synchronization time under PoLiMEr’s power allocation. With each power allocation step, analysis and simulation are brought closer together at time of synchronization.</td>
<td>37</td>
</tr>
<tr>
<td>5.4</td>
<td>Runtime of LAMMPS with PoLiMEr’s power allocation against baseline. Simulation initially assigned more power than analysis.</td>
<td>39</td>
</tr>
<tr>
<td>5.5</td>
<td>Power trace of analysis and simulation running on 64 nodes each, with more power assigned to simulation initially.</td>
<td>40</td>
</tr>
</tbody>
</table>
5.6 Runtime of LAMMPS with PoLiMEr’s power allocation against baseline. Simulation initially assigned 3/4 total nodes at 120 W, and analysis 1/4 of nodes at 120 W per node. ................................. 42
5.7 Power trace of analysis running on 256 nodes and simulation on 768 nodes, each node is set to a 120 W initial power cap. ................................. 43
5.8 Difference between PoLiMEr and baseline in time to reach each synchronization point for one of the 1024-node runs. Before the 260th synchronization step PoLiMEr reaches synchronizations sooner than the baseline. ......................... 44
5.9 Runtime of LAMMPS with PoLiMEr’s power allocation against baseline. Simulation and analysis assigned equal power and nodes. ................................. 45
5.10 Difference between PoLiMEr and baseline in time to reach each synchronization point for one of the 1024-node runs when simulation and analysis start with equal power. ............................................. 46
## LIST OF TABLES

2.1 Power Monitoring and Capping Capabilities on Theta ................................. 9

3.1 Hardware specifications of Theta (XC40) .................................................. 15

3.2 Power and time differences of STREAM using DDR and MCDRAM memory,
    with and without power cap. ................................................................. 21

3.3 Overhead Measurements using DGEMM .................................................... 22

5.1 LAMMPS analysis categories used and their descriptions ......................... 32

5.2 Evaluation Setup Summary. All experiments conducted on Theta .................. 33

5.3 Overview of results when analysis has more power initially ..................... 38

5.4 Overview of results when simulation has more power initially .................. 41

5.5 Overview of results when simulation has more nodes and power initially ....... 44

5.6 Overview of results when simulation and analysis have equal power and nodes. 46
ACKNOWLEDGMENTS

I would like to thank my committee members:

- Hank Hoffmann for being an outstanding academic advisor and mentor and for his
tireless support
- Venkat Vishwanath for being an excellent and supportive supervisor at Argonne
- Ian Foster for his insightful comments and discussions about this project

I would also like to thank the Argonne Leadership Computing Facility for access to computing
resources, and the following scientists and staff for their help, technical support, tutorials
and discussions: Preeti Malakar, Silvio Rizzi, Christopher Knight, Francois Tessier, Sudheer
Chunduri, Benjamin Allen, Derek Jensen, Ryan Milner.
ABSTRACT

The next generation of exascale supercomputing systems is planned to perform a billion billion operations per second to push the frontiers of science. Achieving such compute power raises many new challenges including two addressed by this thesis: operating within strict power limits, and the widening gap between compute and IO speed. Future high performance computing (HPC) systems are expected to handle and maximize performance under strict power limits. Developing software that meets power and energy constraints is a difficult task that must account for dynamic application behavior and system properties simultaneously. Scientific application developers therefore must have a reliable way to measure power and energy consumption and relate these measurements to application- and system-specific events. Currently, scientists face two challenges when measuring and controlling power: (1) diversity—power and energy measurement interfaces differ between vendors—and (2) distribution—power measurements of distributed MPI applications should be unaffected by how a particular applications MPI processes are scheduled. The current state-of-the-art requires scientists interested in power optimization to write tedious, error-prone application- and system-specific code. To make power measurement and management easier for scientists, we propose PoLiMEr, a user-space library that supports fine-grained application-level power monitoring and capping. We evaluate PoLiMEr by deploying it on Argonne National Laboratory’s Cray XC40 Theta system and using it to measure and cap power, scaling the performance and power of several applications on up to 1024 nodes. We find that PoLiMEr requires only a few additional lines of code, but easily allows users to detect energy anomalies, apply power caps, and evaluate Theta’s unique architectural features. The remainder of this thesis relates the second exascale issue – IO speeds falling behind compute speeds — to the issue of power constrained environments. As a result of increasing quantities of data produced by simulations, scientists must turn to online analysis of output data from large-scale simulations. Numerous frameworks and scheduling techniques of simulation and
analysis workloads have been proposed to address this challenge, but none consider the impact of power caps. Using PoLiMEr, we investigate this impact, and propose and implement a power allocation strategy that manages co-dependent executions of simulation and analysis. We evaluate our method on the molecular dynamics simulation LAMMPS on Theta and show we can achieve significant performance improvements of up to $2\times$. 
CHAPTER 1
INTRODUCTION AND MOTIVATION

The race towards exascale computing is in full swing worldwide [11, 18, 37, 47] to achieve compute speeds of a billion billion operations per second, enabling scientific discovery in health sciences, materials and climate science, national security and many more. Achieving such compute power raises many new challenges including two addressed by this thesis: 1) operating within strict power limits, and 2) the widening gap between compute and IO speed [51, 54].

Power is a primary constraint in future exascale systems. The DOE Office of Science estimates that future systems may require around 200 MW to operate [51], which is beyond the reasonably possible, and has set a target of 20 MW for the next generation of supercomputers instead. As such, future systems are expected to operate within strict power constraints, while continuing to meet and exceed high performance guarantees [7, 49]. As power directly impacts performance of systems, software and applications, future supercomputers will need to efficiently manage power as a resource, which is a difficult task due to several reasons, including but not limited to:

- dynamic application behavior — applications exhibit phases that vary in resource usage and have different power and performance characteristics. A good power management mechanism needs to be aware of these phases, as power constraints do not equally affect different resource-specific workloads. Varying power and system characteristics can have different performance impacts on different applications — not all applications necessarily experience a performance boost when given more power, and not all applications suffer drastically from constrained power.
- system heterogeneity and configurability — supercomputers are expected to achieve greater diversity in node-level components and be highly configurable. Some configurations lead to better performance than others, and the configuration spaces may be
• system variability — even in a distributed system consisting of homogeneous components, same workloads on the same nodes may exhibit different performance due to hardware manufacturing variability.

It is evident that both system and application power and performance characteristics need to be well understood to manage power efficiently given the above challenges.

The second exascale issue involves the relatively slow IO speeds not being able to match the advances in compute speed. More compute power enables more sophisticated large-scale scientific applications which in turn may generate data at quantities (terabyte scale [10]) that cannot be efficiently saved to disk or kept in memory. This calls for online, simulation-time analysis, also referred to as in-situ analysis.¹ The HPC community is intensely focused on building frameworks for performing in-situ analysis, e.g., interactive in-situ visualization pipelines, while others focus on how simulation and analysis workflows can be efficiently coupled [14, 29, 31, 32, 39, 40], given IO constraints. Here we consider the problem of running concurrent simulation and analysis workloads given power constraints. While there are many studies evaluating energy and power costs of in-situ analysis methods [2, 3, 15, 19, 22, 48, 58], to the best of our knowledge, in-situ analysis under power constraints hasn’t been given any attention.

To address the performance impacts of in-situ analysis under power constraints, it is crucial to understand the application-level power consumption behavior and impacts of increasing and decreasing power on application events. Such knowledge can be acquired through extensive power and energy profiling of applications. However, that alone is a non-trivial task for scientific application developers. There are two further challenges in current HPC systems that prevent accurate application-level power monitoring: 1) diversity, and 2) dis-

¹The term in-situ analysis has been defined in recent literature with varying degrees of specificity. Some define in-situ analysis more broadly as mentioned here – online analysis of data. Others base the definition on analysis location wherein simulation and analysis share certain resources. In this thesis, we do not assume same resources.
In this work we present PoLiMEr\textsuperscript{2} [34] – a light-weight, scalable and user-friendly application-level API for monitoring and controlling power via targeted power capping. In Chapter 2 we explain the need for PoLiMEr, and provide an overview of its usage and capabilities.

Scientific computing application developers have extensive domain- and application-specific knowledge that PoLiMEr can leverage without requiring developers’ knowledge about power monitoring on their systems. Using PoLiMEr’s abilities to monitor and actuate power we can exploit power and performance optimization opportunities revealed through application behavior. We use these insights to manage in-situ analysis frameworks under power constraints. In Chapter 4 we point out what the power optimization strategies are in a typical in-situ analysis framework. We extend PoLiMEr by implementing a power management mechanism that can decrease and increase power of specific application phases via power capping. PoLiMEr enables us to investigate the impact of power capping in the case of in-situ analysis, and establish a power allocation strategy that manages concurrent executions of simulation and analysis with dependencies between them. We evaluate our method on the molecular dynamics simulation LAMMPS on which we achieve significant performance improvements of up to 2×.
CHAPTER 2

POLIMER: AN ENERGY MONITORING AND POWER
CONTROL API

To develop software that meets power and energy constraints, scientific application developers must have a reliable way to measure these values and relate them to application-specific events. There are two main challenges preventing complete and accurate power monitoring and control in HPC applications: diversity and distribution.

The diversity challenge concerns the growing abundance of power monitoring and control options on current HPC systems. Wallace et al. provide an overview and comparison of vendor supplied environmental power monitoring capabilities, highlighting the differences between vendors in accessing and processing environmental data [55]. This diversity makes taking and interpreting simple power measurements a daunting task to scientists interested in studying or improving their application’s energy efficiency. Interpretation of measurements is further obfuscated by poor documentation of the low-level, vendor-specific power monitoring/control capabilities (e.g. Desrochers et al. note the difficulty understanding Intel’s Running Average Power Limiting (RAPL) interface documentation [12]). Lack of documentation is also true for HPC environments. For instance, while IBM systems have a well documented API for measuring the power of specific hardware components (CPU, memory, optics, PCIe, network transceivers, etc), Cray systems provide a much smaller set of less descriptive power monitoring counters; e.g., their “Energy” counter is not indicative of what it measures at first sight. Measuring application power consumption is thus currently reserved only for expert users well versed in the power characteristics and tools for their specific system.

Even advanced users, however, experience difficulty measuring HPC application power, due to the distribution challenge. This issue concerns the difficulty of obtaining power measurements for applications executed in distributed environments using MPI. Widely used
tools with power measuring capabilities, like PAPI [41] or Linux perf tools [36], do not handle power measurement of parallel applications arbitrarily scheduled on physical nodes. Traditional power profiling tools in an MPI setting add the significant burden of correctly mapping processes to nodes and limit researchers to experimental setups designed for easier power measurement (e.g., always scheduling one process per node). If multiple MPI ranks are scheduled on the same node and read power, users obtain an overestimate of the power consumption. Similarly, if too few processes read power, users may miss some nodes’ power consumption. With the more sophisticated scheduling policies expected from future systems, this problem becomes even worse. Power capping studies face the same issue, and are currently limited to fixed a priori settings given the available tools, providing little flexibility in studying the effects of dynamic power limiting.

To address the diversity and distribution challenges, we propose PoLiMEr—a user-space library for power monitoring and capping of MPI applications. Using known system power interfaces and topology-aware process management together with its easy-to-use API, PoLiMEr enables all HPC users to navigate the power monitoring options on their system, profile large-scale HPC applications and explore power capping effects. PoLiMEr exhibits extremely low overhead and successfully scales to 1024 nodes on Argonne’s system Theta—the cutting-edge supercomputing platform of today.

PoLiMEr fully harnesses a system’s power monitoring and control capabilities by utilizing all available power monitoring and control interfaces and providing an overview of all measurable components without requiring users to know what these power domains are. Simultaneously, through automated process and thread detection, PoLiMEr can profile applications that use any combination of processes and threads. PoLiMEr’s power monitoring and control functionalities are thorough and versatile: it monitors power and energy consumption over time, profiles specific portions of code, sets power caps at any time during application execution and on any combination of nodes. We demonstrate PoLiMEr’s component-wise
power consumption breakdown on Theta, as well as power consumption characteristics and impacts of selective power capping during application runtime on a variety of workloads. Our results demonstrate that PoLiMEr not only provides a wealth of information about an application’s power consumption, but also enables power savings without performance loss by applying power capping during specific application phases.

2.1 Power Monitoring and Limiting in HPC: Background and Related Work

PoLiMEr was inspired by an MPI power profiling application for Blue Gene/Q systems developed at the Argonne Leadership Computing Facility, called MonEQ [56]. PoLiMEr takes power monitoring of MPI applications a step further by adding power capping capabilities, extra features, and customizable power monitoring and capping, while supporting the latest Cray supercomputers.

Several HPC profiling tools solve the diversity challenge, but none simultaneously address the distribution challenge posed by MPI applications. An early evaluation of Argonne’s XC40 system Theta [43], uses PAPI [41] to measure power on a single node. PAPI users, however, are required to know low-level details corresponding to their specific system. Energymon [23] is a portable energy measurement tool that works on a variety of hardware systems. As is the case with PAPI, Energymon has no support for MPI applications. Work such as the Linux “perf” tools [36] can be invoked from a script and wrap an MPI executable as part of job submission. In this case, users must manage MPI process scheduling themselves to obtain an average power consumption per node. Other tools enable collection of out-of-band power information, such as PDUs, the IPMI, PowerInsight [20, 26] or Cray’s Power Management Database (PMDB) [35]. The limitation of script-based approaches and out-of-band tools is the added burden of correctly post-processing the data, mapping readings to corresponding nodes and processes. Furthermore, these data only provide total power measurements,
without any further insight into finer-grained application power characteristics.

Power capping’s impact on applications is increasingly important in the research community. Pedretti et al. show how power capping affects an MPI application [45]. They perform node-level power capping using CapMC on a Haswell-based Cray XC40 system and evaluate its effects across various static P-states. The authors demonstrate poor performance scaling under power caps, even with manual selection of P-states. PoLiMEr offers the ability to selectively apply power caps for specific workloads, which can yield to greater insights into the relationship between performance and power. Lawson et al. demonstrate a runtime model for dynamic power capping on the Xeon Phi [27]. Their experiments show an improvement in energy savings on single-node, multi-threaded benchmarks, suggesting that workload-aware, runtime power-capping can save energy without affecting performance. We believe that PoLiMEr—with its ability to set power caps during application runtime—will further research in this area beyond single-node applications.

Two notable projects take a global perspective on power monitoring and control: Power API [21] and GEOPM [17]. Both aim to standardize system-wide power monitoring and control protocols. PoLiMEr shares similar objectives, however Power API and GEOPM operate within a grand vision to unify applications, schedulers, workload managers, and system software to obtain a holistic power monitoring and control system. PoLiMEr’s goals are comparatively modest, focusing on helping scientists facing the diversity and distribution challenges, by providing an application-level tool that runs at scale with minimal overhead. PoLiMEr is tested at scales up to 1024 nodes on Argonne’s 10 PF Theta system with overheads as low as 0.4%. In comparison, Power API is demonstrated on a single node [27], while GEOPM is evaluated on 12 nodes. In the future, we hope to compare the overheads, capabilities and implications of PoLiMEr, Power API and GEOPM on HPC applications, should these projects become available on Argonne’s systems.
2.2 PoLiMEr: General Framework

This section describes PoLiMEr’s general properties. To better understand PoLiMEr’s features, we explain power monitoring and capping capabilities of Argonne’s XC40 system as a case study, followed by an extensive user guide. Finally, we describe PoLiMEr’s outputs upon finalization and provide an example plot.

PoLiMEr is a user-level library that addresses the power monitoring diversity and the distribution challenge specific to MPI applications. PoLiMEr leverages a system’s power monitoring and capping capabilities without the user needing to explicitly specify what those capabilities are. PoLiMEr provides the necessary implementation abstractions for power monitoring, power capping and MPI process management, which can be used to add support for various systems. Currently, users simply need to compile PoLiMEr for Intel supercomputers with or without the Cray stack. Future releases of PoLiMEr will account for more diverse architectures, including IBM systems, and systems with accelerators. Section 2.3 illustrates how specific power interfaces on Cray’s XC40 are used by PoLiMEr.

Through a topology-aware process organization mechanism, PoLiMEr enables accurate power and energy measurements, reflecting actual utilization of physical nodes and processors, regardless of how the processes or threads are scheduled. It is not affected by user’s process management at the application level, thus making PoLiMEr completely application agnostic. PoLiMEr detects which MPI processes run on which physical nodes using any available processor management interface (e.g., PMI [6]). In the absence of such interface, PoLiMEr uses node hostnames as a fall-back option. The processes are accordingly grouped using communicators specific to each node. One rank per node is then designated as a monitor which reads power and energy, manages PoLiMEr’s internal mechanisms, sets power caps and handles output of collected data. In applications also using OpenMP, PoLiMEr will designate one rank and one thread per node as monitor. PoLiMEr is suitable for large- and small-scale, distributed, parallel or serial applications written in C, Fortran and C++.
Table 2.1: Power Monitoring and Capping Capabilities on Theta

<table>
<thead>
<tr>
<th></th>
<th>Cray</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power interface</td>
<td>pm_counters</td>
<td>RAPL interface</td>
</tr>
<tr>
<td>Power domains</td>
<td>Node, CPU, Memory</td>
<td>Package, Core, DRAM</td>
</tr>
<tr>
<td>Counter types</td>
<td>energy, power</td>
<td>energy</td>
</tr>
<tr>
<td>Update interval</td>
<td>100 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>Power Capping</td>
<td>Node-level using CapMC</td>
<td>Package, Core, DRAM</td>
</tr>
<tr>
<td>CPU Thermal Design Power</td>
<td></td>
<td>215 W</td>
</tr>
</tbody>
</table>

PoLiMEr itself is implemented in C and is available as open source\(^1\).

Using PoLiMEr, a single line of code automates the process of detecting and initializing power monitors. This is a tedious, system- and application-specific process that would otherwise require hundreds of lines of code. Such code would need to be written by an expert user and would not be portable to new systems or even new mappings of processes to nodes on the same system.

### 2.3 Power Monitoring and Limiting on the XC40

PoLiMEr is designed for any Intel-based distributed system with power monitoring and/or power capping capabilities. Here we give a case study of Argonne’s XC40 system Theta to demonstrate the key challenges of diversity of such capabilities. Table 2.1 gives a brief overview of the power monitoring and power capping options on Theta.

On the XC40 Cray provides several out-of-band and in-band power monitoring capabilities [35]. The software-accessible in-band power monitoring capabilities are exposed as unrestricted read-only files called pm_counters, which PoLiMEr uses to obtain power and energy measurements of the node, CPU and memory. These power domains are architected with energy and power counters updated at a frequency of 10 Hz.

Node-level power capping is possible through Cray’s CapMC, an administrative tool.

---

1. https://xgitlab.cels.anl.gov/open-source/PoLiMEr/tags/v1.0
requiring elevated privileges. We designed PoLiMEr to enable power capping studies on applications without needing special privileges via Intel’s RAPL.

RAPL provides energy monitoring and power capping options on the socket level, corresponding to a single Xeon Phi (KNL or Knights Landing) CPU on Theta. RAPL counters are accessible through Model Specific Register (MSR) files with root privileges, however, when available, PoLiMEr utilizes the msr-safe [52] module, which is installed on Theta. Msr-safe allows users without special privileges to access a set of whitelisted MSRs, sufficient for all power monitoring and capping needs.

RAPL records energy readings in designated MSRs, one for package, core or Power Plane 0 (PP0) and DRAM power domain. On Theta, DRAM power capping is not supported, nor is energy monitoring of PP0. Energy can be read for the package and DRAM, while power caps can be set to package and PP0. PoLiMEr determines which MSRs are listed for a particular Intel CPU.

Power limiting on Intel CPUs is performed by maintaining average power consumption over a specified moving time window. RAPL allows for brief violations of the power cap by setting a secondary, shorter time window with a different power cap. Time windows and power caps are configurable by writing to the corresponding MSRs. Reading and writing to the MSRs is handled by any tools accessing RAPL, but there are no application-level power limiting tools. PoLiMEr is a self-contained tool with its own ability to correctly parse, read and set MSR values in MPI applications.

2.4 PoLiMEr Usage

To use PoLiMEr, it suffices to only initialize and finalize it, as shown in Example 2.1. Once initialized, PoLiMEr collects total energy, power and timing information from start to finish. PoLiMEr also polls the power usage at a certain time interval, resulting in power and energy trace information. This interval is set to 0.5 seconds by default.
Example 2.1: Basic usage of PoLiMEr

To profile specific application tasks, users can tag the corresponding code blocks, as shown in Example 2.2. Tags can be arbitrarily named and used in many different combinations.

Example 2.2: Code tagging in PoLiMEr

Tags are reported and time-stamped in the order they are invoked, providing per-call statistics in addition to aggregate information. In other words, users can observe energy, power and time of each call to `inside_loop` tag in Example 2.2, or obtain an average value over all iterations.

Power caps can be set in any arbitrary location in the code with several options. PoLiMEr sup-
plies a default power capping function which is applied to the RAPL package domain, but it also lets users specify further RAPL parameters to set long- and short-term power caps on specific power domains (zones). Likewise, users have the option to retrieve and print power cap values, as well as reset power cap values to default values (PoLiMEr resets power caps on finalization automatically). See Example 2.3.

```c
/* Sets general 150 W long-term and short-term power cap to RAPL package domain */
poli_set_power_cap(150);
/* do work*/

/* Returns current long-term power cap set on RAPL package domain */
double result_watts;
poli_get_power_cap(&result_watts);

/* Setting specific power cap */
poli_set_power_cap_with_params(char *zone_name, double watts_long,
    double watts_short, double seconds_long, double seconds_short);
// zone names in KNL: PACKAGE, CORE, DRAM

/* Getting a specific power cap parameter*/
poli_get_power_cap_for_param("PACKAGE", "watts_long", &result_watts);

/* Resetting power caps*/
poli_reset_system();

/* Printing power cap information*/
poli_print_power_cap_info();

/* Getting maximum and minimum supported power cap*/
double min, max;
```
Example 2.3: Setting power caps in PoLiMEr

PoLiMEr can return the MPI subcommunicator, the node name/ID of the monitoring rank as well as monitor status of any rank. This allows advanced users to apply power caps on a specific set of nodes in a controlled way. Example 2.4 demonstrates how to set power caps on some subset of nodes, where the nodes are labeled numerically on Theta. PoLiMEr will accept any format of node labels.

Example 2.4: Setting power caps on some select nodes

```c
int node_num;
poli_get_monitor_node(&node_num);
if (node_num > 3000)
poli_set_power_cap(150);
```

2.5 PoLiMEr Output

After PoLiMEr is finalized, a set of three output files will be produced for each node the application used:

- PoLiMEr_node-id_job-id.txt - contains the polled power measurements over time (power trace), marking the start and end of all user tags and power caps
- PoLiMEr_energy-tags_node-id_job-id.txt - contains a summary for all user tags, displaying total energy, power and time of the tagged code blocks. This file also includes a special, generated tag called application_summary which provides total energy, time and power from when PoLiMEr was initialized to when it was finalized
- PoLiMEr_powercap-tags_node-id_job-id.txt - contains records of when and what power caps were set

PoLiMEr comes with a supplementary Python script that can process these files as tab-separated CSV files (they can be read by any CSV-file processing tools). Furthermore,
setting of the PoLi_PREFIX environment variable allows users to specify additional naming or location of the output files. All files contain time, power and energy specific columns. The time columns track time stamps, elapsed times since initialization or between two points in time. The power and energy columns record power and energy measurements for the RAPL zones and additional system-specific counters (e.g. Cray counters on the XC40). The trace file also records the polled power caps for each time interval, while the power cap tag file lists all user (energy) tags that were active during a particular power cap.

Figure 2.1 shows the output of the trace file of an idle node running an MPI program that sleeps for a minute. On Theta we can measure power consumption of the package and DRAM RAPL domain, as well as access the Cray power counters for node, CPU and memory. A user would see this breakdown of components without explicitly specifying them. The RAPL package domain and Cray CPU counter correspond in measurements, as is the case with RAPL DRAM and Cray memory. The Cray node power counter measures power draw of other peripheral components such as the on-board SSD and the NIC.
CHAPTER 3
POLIMER EVALUATION

In this section we demonstrate the plethora of insights PoLiMEr can reveal. We show how PoLiMEr can identify application phases and optimization targets for achieving energy or performance goals on real simulation code (LAMMPS) at scale up to 1024 nodes. We demonstrate PoLiMEr’s ability to set power caps at select phases during application runtime, and show how Theta’s architectural features, specifically memory configuration, can benefit from power capping. Finally, we present overhead measurements of PoLiMEr. The experiments were run on Theta. Table 3.1 contains an overview of Theta’s specifications.

3.1 Characterizing Application Power Consumption Behavior using PoLiMEr

Here we give an overview of the types of insights into application power consumption characteristics PoLiMEr can provide using LAMMPS [46], a molecular dynamics simulation, as a case study. The experiments presented here use the default scaled Rhodopsin benchmark that is supplied with the source code [25], which allows users to specify the system dimensions (also referred to as repetitions in the benchmark setup file) and number of steps or iterations. On Theta, LAMMPS is compiled with the USER-INTEL and USER-OMP packages. We

| Table 3.1: Hardware specifications of Theta (XC40) |
| --- | --- |
| # Nodes | 4,392 |
| CPU | 7230 KNL |
| # Cores | 64 |
| # nodes/rack | 192 |
| L1 cache | 32 KB instr. + 32 KB data |
| L2 cache | 1 MB |
| High-bandwidth memory | 16 GB |
| Main memory | 192 GB |
| NVRAM/node | 128 GB SSD |
| Interconnect | Cray Aries Dragonfly |
Figure 3.1: LAMMPS Power Trace. 128 nodes, 64 ranks/node, 2 threads/rank. Rhodopshin benchmark with $6 \times 6 \times 6$ repetitions (6.9 million atoms total) and 100 iterations. Used the flat memory mode and the quad cluster mode on KNL.

Figure 3.1 shows the power trace obtained from running LAMMPS on 128 nodes. Approximately the first half of the runtime was spent in the initialization phase. The simulation was performed in 100 iterations. The main simulation phases in LAMMPS can be identified in the verlet.cpp source file, which builds Verlet neighbor lists and runs the key components of the simulation. We used PoLiMEr’s tagging feature on this file to obtain energy values for some of LAMMPS’s key phases, including setup and building of the neighbor list (neighbor_setup, neighbor_build), pair computation (force_pair), Kspace computation (force_kspace), atom property computation (force_atom), Verlet cleanup, and writing of output. PoLiMEr’s energy tags file lists tags as they were called, which allows for easy counting of how many times a particular code block was executed. In the 100 iterations, neighbor setup, atom, Kspace and pair computations are executed at each iteration. Neighbor build is performed 11 times, writing of output twice, and the cleanup phase is called once. Figure 3.3 shows the average energy consumption and time per tag per iteration. The Kspace computation and neighbor build are the most energy consuming phases, yet their energy consumption is not proportional to the duration of these phases, as shown in Figure 3.3. Thus, PoLiMEr’s tagging feature enables us to identify potential optimization strategies with minimal changes to the...
Figure 3.2: Average energy consumption per tag per iteration.

Figure 3.3: Average time per tag per iteration.

Figure 3.4: Energy and time values per tag per iteration for LAMMPS on 128 nodes.
Figure 3.5: 128 Nodes

Figure 3.6: 1024 Nodes

Figure 3.7: LAMMPS total energy consumption and time per node for two runs using 128 and 1024 nodes respectively.

source code (a few tags in a single source file): to optimize for time, the neighbor build phase needs to be addressed, and the Kspace computation to optimize for energy.

In addition to user specified tags, PoLiMEr will generate an application_summary tag, used to obtain total application energy, power and time measurements per node. Figure 3.7 shows the total time and energy per node for two separate runs of LAMMPS on 128 nodes and on 1024 nodes. Each of the data points represents a single node. From Figure 3.7 it is evident that some nodes are more time efficient, others more energy efficient. In the 128-node run there is a $4 \times$ performance range, while in the large-scale run there is a difference in energy consumption of close to 20%. Since PoLiMEr’s output files are labeled per node, it is easy to identify the exact node label for each of these data points. Therefore, PoLiMEr can enable future research on node variability in energy consumption.

### 3.2 Power Capping with PoLiMEr

We demonstrate PoLiMEr’s capability to set power caps during application runtime using Intel’s MKL DGEMM (1 rank/node, 64 threads/rank, 1 thread/core). Power caps were set on the RAPL package domain using PoLiMEr’s default power capping function.
Figure 3.8: Intel MKL DGEMM executed on a single node, 1000 iterations (1024 × 1024 matrix size). RAPL package power cap was increased every 100 iterations by 20 W, starting with 100 W. Vertical lines mark increase in power cap.

Figure 3.9: No power cap

Figure 3.10: 110 W RAPL package power cap

Figure 3.11: Power consumption trace of STREAM using DDR memory on 64 nodes (1 MPI rank/node, 64 threads/rank, 1 thread/core), without and with power cap. Flat memory mode, quad cluster mode on KNL. 15 GB of memory, 100 iterations.
We used a matrix size of $1024 \times 1024$, and DGEMM was run for 1000 iterations. An increasing power cap was set every 100 iterations, and the power trace is presented in Figure 3.8. As a compute bound benchmark, DGEMM is very susceptible to changes in performance with changing power caps, each time consuming as much power as it was allowed to. The system’s responsiveness to changes in power cap is fairly quick, taking roughly a second for changes in power consumption to be observable.

### 3.3 Power Capping with PoLiMEr: An Architectural Perspective

Power and performance impacts under power cap are not only application-dependent, but also system dependent. Here we use PoLiMEr to explore differences in power and performance between Theta’s in-package (MDRAM) and external (DDR) DRAMs under power cap. We ran STREAM on 64 nodes (1 rank/node, 64 threads/rank, 1 thread/core), using the flat memory and quadrant cluster mode. Each instance used 15 GB of memory, and the four STREAM benchmarks, copy, add, scale and triad were executed in a loop of 100 iterations.

Figure 3.9 shows the power consumption over time for STREAM using the slower (90 GB/s) but larger DDR memory. Table 3.2 shows the power and time difference between DDR and the faster (480 GB/s) MCDRAM. Each node consumed on average 120 W - 140 W during the main benchmark phase when using DDR memory, and around 200 W in the MCDRAM case.

Figure 3.10 shows the total power consumption over time for STREAM on DDR with a 110 W power cap, with a decrease in node and package power consumption. The memory power consumption was unaffected, and all four benchmarks finished equally fast in both the capped and the uncapped version. In contrast, STREAM’s verification phase past the 70 second mark is significantly impacted by the power cap. The reported bandwidth in the uncapped version for copy and scale was 82 GB/s, and 87 GB/s for add and triad. In the
Table 3.2: Power and time differences of STREAM using DDR and MCDRAM memory, with and without power cap.

<table>
<thead>
<tr>
<th></th>
<th>DDR</th>
<th>MCDRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time for main loop</td>
<td>70 s</td>
<td>18 s</td>
</tr>
<tr>
<td>Max Node Power</td>
<td>14 kW</td>
<td>18 kW</td>
</tr>
<tr>
<td>Max Package Power</td>
<td>9 kW</td>
<td>14 kW</td>
</tr>
<tr>
<td>Max Memory Power</td>
<td>2 kW</td>
<td>1 kW</td>
</tr>
<tr>
<td>Time for main loop w/ 110 W cap</td>
<td>70 s</td>
<td>39 s</td>
</tr>
<tr>
<td>Max Node Power w/ 110 W cap</td>
<td>12 kW</td>
<td>11 kW</td>
</tr>
<tr>
<td>Max Package Power w/ 110 W cap</td>
<td>7 kW</td>
<td>7 kW</td>
</tr>
<tr>
<td>Max Memory Power w/ 100 W cap</td>
<td>2 kW</td>
<td>1 kW</td>
</tr>
</tbody>
</table>

capped version, copy and scale bandwidth was 81 GB/s and 86 GB/s for add and triad. These results suggest that the package power cap had little to no impact on STREAM’s performance when using the DDR memory. On the other hand, there is a performance penalty in the IPM case, as can be seen in Table 3.2.

3.4 Overhead Measurements

We present overhead measurements of PoLiMEr below. We ran DGEMM on 16, 128 and 512 nodes with PoLiMEr, and timed PoLiMEr’s essential features. The results were reported for 1 rank and are displayed in Table 3.3. Initialization and finalization time increase with increasing number of nodes. Finalization also depends on the number of user tags, as PoLiMEr has to process all tags, check they have been completed correctly, and write them out to file.

A tag may be used multiple times if it is inside a loop, for instance. We emulated this behavior by measuring the overhead of starting and ending a tag 100 times, and are reporting the overhead of finalizing PoLiMEr in that case. Tagging incurs relatively low overhead, while finalization scales with the number of tags used. The overhead of setting and getting power caps is near negligible, so users are free to set and poll power caps as many times as they wish. The overhead to poll the system 1000 consecutive times shows there is close to no cost of polling the system regardless of the polling interval. Any actions
Table 3.3: Overhead Measurements using DGEMM

<table>
<thead>
<tr>
<th># Nodes</th>
<th>16</th>
<th>128</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>application time (s)</td>
<td>192.103</td>
<td>191.56</td>
<td>201.410</td>
</tr>
<tr>
<td>initialize (s)</td>
<td>0.061</td>
<td>0.098</td>
<td>0.201</td>
</tr>
<tr>
<td>finalize - 2 tags (s)</td>
<td>0.092</td>
<td>0.209</td>
<td>0.84</td>
</tr>
<tr>
<td>start 1 tag (s)</td>
<td>0.004</td>
<td>0.003</td>
<td>0.003</td>
</tr>
<tr>
<td>end 1 tag (s)</td>
<td>0.0003</td>
<td>0.0003</td>
<td>0.0003</td>
</tr>
<tr>
<td>start + end tag 100× (s)</td>
<td>3.361</td>
<td>3.539</td>
<td>2.015</td>
</tr>
<tr>
<td>finalize w/ 100 tags (s)</td>
<td>18.531</td>
<td>23.467</td>
<td>38.639</td>
</tr>
<tr>
<td>set power cap 100× (s)</td>
<td>0.002</td>
<td>0.002</td>
<td>0.001</td>
</tr>
<tr>
<td>get power cap 100× (s)</td>
<td>0.0004</td>
<td>0.0005</td>
<td>0.0005</td>
</tr>
<tr>
<td>polling overhead 1000× (s)</td>
<td>0.00048</td>
<td>0.00042</td>
<td>0.00041</td>
</tr>
</tbody>
</table>

performed by PoLiMEr during the main execution of the application incur near negligible cost. The heavier initialization and finalization phases are outside of the main application tasks, thus not affecting the outcomes of the application.
CHAPTER 4
POWER ALLOCATION IN IN-SITU ANALYSIS

In-situ analysis is a mode of processing data from scientific computing simulations at simulation time. It allows scientists to launch calculations and/or to visualize simulation data as it becomes available in order to monitor progress of the simulation. The overall objective of any in-situ analysis framework is to run as many analyses as possible to extract as much valuable information about the simulation. However, more frequent analysis requires more resources which are limited by IO speeds and power. In the following section, we provide an overview of prior work that addresses the IO limitation along other related past research. In Section 4.2 we describe the general flow of computation in a typical in-situ analysis framework, and show where power optimization strategies lie when such frameworks have to operate within a power limit. Finally we present an application-aware power management strategy that reallocates the available power budget based on the simulation and analysis needs.

4.1 Related Work

Many in-situ analysis frameworks have been built to date [1, 5, 8, 13, 14, 16, 24, 28, 42, 59, 60], focusing on the efficacy of this mode of simulation-time analysis. These frameworks aim to enable scalable and fast in-situ analysis with low synchronization overhead between simulation and analysis. However, few consider resource constraints. Malakar et al provide several studies [30–32] that consider IO and memory constraints. These studies consider a set of analyses with different importance and resource requirements, and aim to schedule the most important analyses as many times as possible. They formulate this as a constrained optimization problem, and provide an offline method for finding a schedule that determines the frequency of each desired analysis. Even though their formulation can be easily extended to include power as a constraint, extensive profiling would be required before being able to run
the in-situ framework with the optimal schedule of analyses. In this thesis, we seek to provide an online solution that does not require profiling of power and performance requirements by the user.

While in-situ analysis under power constraints hasn’t been extensively studied yet, there are several works that recognize the importance of characterizing energy and power requirements of in-situ analysis frameworks. Some [15, 19, 22, 48, 58] characterize and provide energy and power models for estimating energy costs associated with different I/O optimization schemes. Adhinarayanan et al [2, 3] compare the energy cost of in-situ analysis against post-processing mode and find that in-situ analysis and visualization, in addition to mitigating issues with IO constraints, also lowers energy consumption. Being able to estimate power and energy costs of in-situ analysis frameworks is important in evaluating the power-and energy-efficiency of how these frameworks couple simulation and analysis, however we are the first to propose a strategy for managing power for such frameworks.

Power as a constraint in HPC has been the topic of many studies tackling the problem at the hardware and system level. Some of these efforts are discussed in Section 2.1. Several additional works [9, 44, 61] consider system-wide power management in power-constrained systems at the job scheduler level. Others [33, 57] consider power management at the MPI-task granularity. These approaches do not generalize well to in-situ analysis frameworks, and cannot take advantage of the unique power optimization opportunities exposed through application behavior. Sundriyal et al [53] do work with in-situ simulation and analysis but their power management strategy is a runtime approach that shifts power between different RAPL domains, and do not consider any application-level information. Savoie et al [50] exploit slack between communication-bound and compute-bound phases of applications at the task level and demonstrate results on simulated experiments. In contrast, we propose a power management strategy that leverages slack as a result of the specific structure that in-situ analysis frameworks exhibit and demonstrate results on real workloads at scale.
4.2 The Structure of In-situ Analysis and Impacts of Power Constraints

The case of in-situ analysis can be generalized as concurrently running a set of workloads that assume the role of the analysis and a set of workloads part of the simulation. The two sets communicate with each other periodically, as the analysis requires updated output from simulation to perform some computation on the obtained data. For instance, in in-situ interactive visualization, the simulation consists of some scientific simulation driving scientific objectives, and the analysis consists of graphics rendering tasks and handling of user interaction. In other cases, analysis may be a set of calculations that extract certain information from the simulation of interest to the scientists.

Here we consider the problem of running the simulation on one set of nodes, and the analysis on a separate set of nodes, which is a common setup especially with remote in-situ analysis. When analysis queries the simulation for data and simulation responds, both workloads enter a synchronization phase where the data is sent. The synchronization indicates a dependency between the two workloads. Once the data has been sent and received, both simulation and analysis continue to carry out their respective computations independently of each other. We label this set of concurrent and independent tasks between two synchronization points as compute phases. In general, in-situ analysis frameworks consist of alternating compute and synchronization phases and there are a total of $K$ compute steps.

During a particular compute step simulation and analysis may have different amount of work to complete before they can synchronize. Furthermore, under a power constraint, there is some initial distribution of the total power budget to the respective node partitions for the two workloads. Given this power allotment and the amount of work to be completed by each, the two may not reach the synchronization point at the same time. The faster one will have to wait for the other to reach the synchronization point, and its nodes will idle during that time, ultimately wasting unused power. This creates an opportunity to optimize
Figure 4.1: Diagram of power consumption over time of a particular compute step between two synchronization points. The red line indicates the global power budget. On the left, analysis finishes sooner and waits for the simulation to reach the synchronization point. On the right, power is reallocated from analysis to speed up simulation such that both can synchronize sooner than with the initial power distribution.

power utilization, in which power could be reallocated to speed the slower workload up, and slow the faster one down such that they reach the synchronization point at the same time. See Figure 4.1. As a result, the synchronization point would be reached sooner than with the initial power distribution. Reallocating power in this way for every compute step would reduce the runtime of each successive step, thereby reducing the overall runtime and enabling more analysis.

There exists some optimal distribution of power between simulation and analysis such that no power is wasted when the two need to synchronize. This optimal distribution is unknown, but we aim to converge to the optimal from an initial power distribution. The following section formally describes how we can converge to the optimal distribution.

4.3 Power Allocation Between Simulation and Analysis

The problem of power-constrained in-situ analysis can be formalized as follows. Let $K$ be the total time steps for which the simulation has to run. The simulation steps consist
of interleaved synchronization steps during which simulation and analysis communicate, and compute steps during which simulation and analysis perform independent work. The synchronization and compute steps alternate.

Let $T_S(k)$ and $P_S(k)$ be the runtime and total power consumption of simulation during compute step $k$ which occurs between two synchronization steps. Similarly, let $T_A(k)$ and $P_A(k)$ be the runtime and power of the analysis during time step $k$.

We want to minimize the time it takes to reach each synchronization point:

$$\forall k \text{ minimize } \max \{T_S(k), T_A(k)\}$$

The optimal solution to this optimization problem is when $T_S(k) = T_A(k)$. *Proof:* Suppose the optimal solution is achieved when two intervals $T_S(k)$ and $T_A(k)$ are not equal. If $T_S(k) > T_A(k)$, then the $T_S(k)$ can be made shorter to match $T_A(k)$ which leads to a better solution of $T_A(k)$ length, which contradicts the assumption that the runtime is minimized when $T_S(k) > T_A(k)$. The same applies if we assume that $T_A(k) > T_S(k)$. Therefore, the two intervals must be equal.

Let $P_{S OPT}(k)$ and $P_{A OPT}(k)$ be the optimal power consumption associated with simulation and analysis when $T_S(k) = T_A(k)$.

Given a global power limit $C$, we also have the following constraint:

$$P_S(k) + P_A(k) \leq C$$

The principle behind our approach is to at any time step $k$ assign a portion $r_k$ of the power budget $C$ to the simulation and the remainder $(1 - r_k)C$ to the analysis, such that their runtimes are equal for that time step. In other words:

$$P_{S OPT}(k) = r_k C$$

$$P_{A OPT}(k) = (1 - r_k)C$$

Therefore:

$$P_{S OPT}(k) + P_{A OPT}(k) = C, \forall k$$
We start by assuming a relationship between power and runtime of the form for every
time step $k$:

$$
\frac{1}{T_S(k) \times P_S(k)}, \quad \frac{1}{T_A(k) \times P_A(k)}
$$

Then we solve the following system of equations:

$$
\alpha_S(k)T_S(k) = \alpha_A(k)T_A(k)
$$

$$
P^{OPT}_S(k) + P^{OPT}_A(k) = C
$$

Which yields the following optimal power allocations:

$$
P^{OPT}_S(k) = \frac{\alpha_A(k)}{\alpha_S(k) + \alpha_A(k)}C,
\quad P^{OPT}_A(k) = \frac{\alpha_S(k)}{\alpha_S(k) + \alpha_A(k)}C
$$

We set the following ratio:

$$
r_k = \frac{P^{OPT}_S(k)}{C}
$$

Note that $\alpha_S(k)$ and $\alpha_A(k)$ originate from an assumed relationship between time and
power depending on their actual measurements. The true underlying power and performance
function depends on the workload, the application’s resource needs and system configuration,
however a linear relationship between power and performance is a suitable representative of
typical HPC workloads. As the underlying power and performance relationship is unknown,
we choose an exponentially weighted moving average to leverage information from recent
measurements:

$$
P^{alloc}_S(k) = (1 - r_k)P^{OPT}_S(k - 1) + r_kP^{OPT}_S(k)
$$

$$
P^{alloc}_A(k) = (1 - r_k)P^{OPT}_A(k - 1) + r_kP^{OPT}_A(k)
$$

Allocating power according to Equation 4.3 assumes that both workloads operate at the
power cap already. The case when the workloads are not both at the power cap will be
addressed in the future work, and the extension is simple: shift allocated power from the
low-power task to the nodes running the other task. Given that future systems are expected
to operate under strict power caps, it is reasonable to assume that many HPC workloads
will operate close to the power cap. As such handling the case where both simulation and analysis run at the power limit is a more pressing issue.

Furthermore, these power allocations are set via power capping, so another implicit assumption is that workloads will operate at the power we set. However, these power settings have to be within certain physical limits of the system. There is a minimum amount of power required for any workload to run at the slowest possible speed, and each workload has a maximum speed it can achieve under some power setting. Finding these workload-dependent bounds also remains to be addressed in future work, but currently we can perform an extra adjustment after $P_{alloc}^S(k)$ and $P_{alloc}^A(k)$ are determined, such that we don’t operate below the machine’s operating power and we don’t exceed the Thermal Design Power of the machine. Therefore, we perform an extra adjustment by $\delta_k \geq 0$ if the allocated powers are outside of desired bounds:

$$P_{alloc}^S(k) = (1 - r_k)P_{OPT}^S(k - 1) + r_kP_{OPT}^S(k) + \delta_k$$

$$P_{alloc}^A(k) = (1 - r_k)P_{OPT}^A(k - 1) + r_kP_{OPT}^A(k) - \delta_k$$

Our power allocation method does not violate the global power constraint $C$. Proof: At any time step $k$ the total power assigned is:

$$P_{alloc}^S(k) + P_{alloc}^A(k) = (1 - r_k)P_{OPT}^S(k - 1) + r_kP_{OPT}^S(k) + \delta_k + (1 - r_k)P_{OPT}^A(k - 1) + r_kP_{OPT}^A(k) - \delta_k = (1 - r_k)\left(P_{OPT}^S(k - 1) + P_{OPT}^A(k - 1)\right) + r_k\left(P_{OPT}^S(k) + P_{OPT}^A(k)\right) = (1 - r_k)C + r_kC = C$$

To summarize, our power allocation method works as follows.

At the beginning of every synchronization step $k$:

1. obtain measurements of $T_S(k)$, $T_A(k)$, $P_S(k)$ and $P_A(k)$ from the corresponding compute step $k$

2. use these measurements to obtain $\alpha_S(k)$ and $\alpha_A(k)$ according to Equation 4.1

3. compute $P_{OPT}^S(k)$ and $P_{OPT}^A$ according to Equation 4.2
4. obtain new power allocations using Equation 4.3
5. determine $\delta_k$ by checking if reallocated power is outside of physical range
6. divide the new allocated powers for simulation and analysis evenly among their respective nodes, and set a power cap to each node

Note that the last step leaves room for more complex power distribution within the nodes running simulation and analysis. We choose an even distribution because high-performance simulations tend to have high degrees of parallelism, distributing workloads evenly among nodes.
CHAPTER 5
EVALUATION OF IN-SITU POWER ALLOCATION

We extend PoLiMEr to perform power reallocation between synchronization points as described in Section 4.3 and test it on LAMMPS. The following sections describe LAMMPS and the evaluation setup.

5.1 Integrating PoLiMEr with LAMMPS

LAMMPS [25, 46] is an open-source massively parallel molecular dynamics simulation, and representative of many typical HPC workloads. In this work we use LAMMPS version February 2018. It is a versatile, modular software written in C++ that can be used as a library or a stand-alone application. It can simulate different systems (liquids, solids, etc) containing different types of particles (atoms, molecules, ions, etc), which are divided into sub-volumes assigned to individual MPI ranks. Further multithreaded parallelization can be used to apply molecular dynamics computations on these sub-volumes, such as integrating Newton’s laws of motion.

Different available timestepping algorithms dictate the progress of simulation, and here we consider the velocity-Verlet timestepping algorithm. The Verlet algorithm can invoke optional computations – fixes and computes in LAMMPS terminology – at the end of each time step, which is how a set of analyses can be invoked during the simulation. Malakar et al [30, 31] propose an extension to the Verlet algorithm called Verlet-Splitanalysis which enables many configurations of the LAMMPS simulation and LAMMPS analyses that dictate which nodes run the simulation and analysis. This extension creates an MPI subcommunicator for every simulation-analysis pair of nodes. We use the Splitanalysis extension to run simulation and analysis on separate sets of nodes while varying the partition size of the nodes.

The particular LAMMPS simulation considered here is a system of water molecules sol-
Table 5.1: LAMMPS analysis categories used and their descriptions.

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hydronium RDF</td>
<td>Compute hydronium-water, hydronium-hydronium, and hydronium-ion radial distribution function (RDF), averaged over all molecules</td>
</tr>
<tr>
<td>ion RDF</td>
<td>Compute ion-water and ion-ion RDFs averaged over all molecules</td>
</tr>
<tr>
<td>VACF</td>
<td>Compute velocity auto-correlation function (VACF) for water-oxygen, hydronium-oxygen, and ion atoms</td>
</tr>
<tr>
<td>MSD</td>
<td>Compute mean squared displacements (MSD) for 1D and 2D spatial bins, and averaged over all hydronium and ions</td>
</tr>
</tbody>
</table>

Vating two types of ions, and we consider 4 categories of analyses commonly used in scientific computing [4, 38], used as case studies in [30–32] and summarized in Table 5.1. Each of these analyses have different resource usage, e.g., MSD has high memory and CPU utilization, MSD1D and MSD2D have mostly high memory utilization, while RDF is more compute bound [31]. Every analysis can be invoked at a specified frequency of simulation steps. We chose a frequency of 2 for RDFs and VACF, and a frequency of 20 for the MSDs, which effectively means that simulation and analysis will synchronize every 2 simulation steps. At every synchronization step, simulation sends particle coordinates to the analysis. By having different frequencies for the analyses, we can emulate varying workloads on the analysis node partition.

To integrate Verlet-Splitanalysis with PoLiMEr only a few extra lines of code are required. We first group the ranks handling simulation and analysis and pass their MPI communicators to PoLiMEr. PoLiMEr takes an MPI Group intersection with its own monitor ranks to determine which monitor ranks are simulation ranks and which are the analysis ranks. In our implementation this took one line of code to pass communicators to PoLiMEr and 4 extra lines in LAMMPS to group the ranks into simulation and analysis ranks. If simulation and analysis were two separate MPI applications, they can be launched via the MPI multi-launch method to access the communicators. Finally, we surround the synchronization calls in LAMMPS by calls to PoLiMEr indicating the beginning of the synchronization step and the end of the synchronization step.
Table 5.2: Evaluation Setup Summary. All experiments conducted on Theta.

<table>
<thead>
<tr>
<th>LAMMPS problem description</th>
<th>box of water molecules solvating two types of ions</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAMMPS problem size</td>
<td>27.5 million particles</td>
</tr>
<tr>
<td>Number of Nodes</td>
<td>64, 128, 512, 1024</td>
</tr>
<tr>
<td>Ranks/Node</td>
<td>64</td>
</tr>
<tr>
<td>Threads/Rank</td>
<td>2</td>
</tr>
<tr>
<td>Total simulation steps</td>
<td>1000</td>
</tr>
<tr>
<td>Analysis frequency</td>
<td>MSDs every 20, RDFs &amp; VACF every 2 steps</td>
</tr>
<tr>
<td>PoLiMEr power allocation frequency</td>
<td>every 20 synchronization steps (40 simulation steps)</td>
</tr>
</tbody>
</table>

Furthermore, PoLiMEr has a configurable parameter that determines after how many synchronization steps power should be reallocated. This is useful when simulation and analysis synchronize frequently. In a sample run of LAMMPS with the analyses in Table 5.1 on 512 nodes under a global power budget of $512 \times 120$ W, the mean time between two synchronization points was 0.33 seconds (0.29 seconds median time) out of 500 synchronizations in total. Reallocating power at that frequency may be influenced by noise, so we set a frequency of every 20 synchronization points to reallocate power.

## 5.2 Evaluation Setup and Results

We conduct a sequence of experiments where we assign a global power budget to LAMMPS and its analyses, and evaluate PoLiMEr’s power allocation strategy against the baseline, which is the default RAPL and job scheduler behavior. The baseline reflects current mode of operation where systems operate under a desired power cap divided equally between all nodes. We run LAMMPS with and without PoLiMEr on 64, 128, 512 and 1024 nodes. To minimize the effect of node hardware variability, we run PoLiMEr and baseline on the same set of nodes. This pair of runs is repeated 3 times for 64, 128, 512, 1024 nodes each, and for each node count we report the median runtime from the 3 runs. Table 5.2 provides an overview of LAMMPS-specific parameters.

We consider three main types of initial power distributions:

1. analysis assigned more power than simulation
2. simulation assigned more power than analysis
3. analysis and simulation assigned equal amount of power

In a typical LAMMPS run with the parameters chosen for these experiments, 80 – 90 W per node is enough power to achieve a stable state. If the power cap is below this threshold, some bursty power consumption behavior may occur in which the power cap is briefly violated. Similarly, when allowed to run up to the Thermal Design Power, LAMMPS reaches power consumption at around 160 W per node for the majority of its running time. Therefore, we chose a 120 W power cap per node to emulate a moderately power-constrained environment, and to leave sufficient room for power to be shifted around.

5.2.1 Initial power distribution: analysis has more power

Here analysis and simulation run on equal amount of nodes. For instance, in the 64-node experiment analysis and simulation run on 32 nodes each. Before the Verlet-Splitanalysis timestepping algorithm is invoked and the simulation-analysis context is created, LAMMPS goes through an initial setup phase which we cap at 120 W per node. Once Verlet-Splitanalysis is invoked, the analysis nodes start at 140 W per node, and the simulation nodes start at 100 W. Figure 5.1 shows the median total runtime of repeated LAMMPS runs with and without PoLiMEr.
Figure 5.1: Runtime of LAMMPS with PoLiMEr’s power allocation against baseline. Analysis initially assigned more power than simulation.

With 58% power of the global power budget assigned to analysis, PoLiMEr is able to significantly reduce the runtime by allocating power to a different power distribution compared to the baseline that maintains this distribution. Up to 512 nodes there is $50\% - 52\%$ runtime reduction and $43\%$ reduction on 1024 nodes. Figure 5.2 shows the power trace on one of the 64-node runs and demonstrates that PoLiMEr shifts power from the analysis to simulation at each power allocation step.
Figure 5.2: Trace of power consumption of analysis and simulation running on 32 nodes each. PoLiMEr reallocates power every 40 simulation steps, gradually increasing performance compared to the baseline that maintains initial power allocation. Time and power axis are at the same scale.

Figure 5.3 shows the time to reach synchronization by simulation and analysis in the same 64-node run with PoLiMEr. Since PoLiMEr’s power allocation frequency is set to every 20 synchronization steps, the first power reallocation occurs at the 20th synchronization step. At this step, the analysis was faster than the simulation. With each power allocation step, the time difference between analysis and synchronization reduces.
Figure 5.3: Difference between analysis and simulation in time to reach synchronization time under PoLiMEr's power allocation. With each power allocation step, analysis and simulation are brought closer together at time of synchronization.

Table 5.3 contains a summary of all runs for this set of experiments where analysis was assigned more initial power.
Table 5.3: Overview of results when analysis has more power initially.

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>PoLiMEr Time (s)</th>
<th>Baseline Time (s)</th>
<th>Final PoLiMEr %C to simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>1072.11</td>
<td>2303.12</td>
<td>51.78%</td>
</tr>
<tr>
<td>64</td>
<td>1094.62</td>
<td>2108.62</td>
<td>50.57%</td>
</tr>
<tr>
<td>64</td>
<td>1319.10</td>
<td>3650.28</td>
<td>51.44%</td>
</tr>
<tr>
<td>128</td>
<td>643.65</td>
<td>1330.15</td>
<td>49.05%</td>
</tr>
<tr>
<td>128</td>
<td>653.64</td>
<td>1108.79</td>
<td>48.36%</td>
</tr>
<tr>
<td>128</td>
<td>726.13</td>
<td>1809.62</td>
<td>49.39%</td>
</tr>
<tr>
<td>512</td>
<td>323.18</td>
<td>608.99</td>
<td>45.28%</td>
</tr>
<tr>
<td>512</td>
<td>350.78</td>
<td>787.66</td>
<td>45.95%</td>
</tr>
<tr>
<td>512</td>
<td>316.66</td>
<td>532.53</td>
<td>44.78%</td>
</tr>
<tr>
<td>1024</td>
<td>451.44</td>
<td>1151.81</td>
<td>45.51%</td>
</tr>
<tr>
<td>1024</td>
<td>402.91</td>
<td>669.68</td>
<td>43.90%</td>
</tr>
<tr>
<td>1024</td>
<td>357.23</td>
<td>607.85</td>
<td>43.93%</td>
</tr>
</tbody>
</table>

5.2.2 Initial power distribution: simulation has more power

In this round of experiments we schedule simulation and analysis on equal amount of nodes, but we start the simulation at 140 W per node and the analysis at 100 W per node once Verlet-Split analysis is invoked. Figure 5.4 gives an overview of the runtimes observed when simulation has more power initially.
Figure 5.4: Runtime of LAMMPS with PoLiMEr’s power allocation against baseline. Simulation initially assigned more power than analysis.

Assigning more power to simulation is how we would expect scientific application developers to configure the power budget, as simulations are typically more compute-intensive. PoLiMEr still achieves significant results, improving performance by 35% on 64 nodes, 42% on 128 nodes, 27% on 512 nodes and 22% on 1024 nodes.
Figure 5.5 shows the power trace of one of the 128-node runs. This result shows that PoLiMEr can allocate power from simulation to analysis to improve overall performance. Table 5.4 contains a detailed overview of the results in the case where simulation starts with 58% of the global power budget.
Table 5.4: Overview of results when simulation has more power initially.

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>PoLiMEr Time (s)</th>
<th>Baseline Time (s)</th>
<th>Final PoLiMEr %C to simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>784.64</td>
<td>1225.61</td>
<td>51.69%</td>
</tr>
<tr>
<td>64</td>
<td>834.11</td>
<td>1219.13</td>
<td>52.56%</td>
</tr>
<tr>
<td>64</td>
<td>790.12</td>
<td>1164.11</td>
<td>52.35%</td>
</tr>
<tr>
<td>128</td>
<td>520.13</td>
<td>902.64</td>
<td>50.19%</td>
</tr>
<tr>
<td>128</td>
<td>527.14</td>
<td>850.12</td>
<td>50.81%</td>
</tr>
<tr>
<td>128</td>
<td>499.65</td>
<td>942.62</td>
<td>49.65%</td>
</tr>
<tr>
<td>512</td>
<td>268.62</td>
<td>355.37</td>
<td>45.98%</td>
</tr>
<tr>
<td>512</td>
<td>281.86</td>
<td>396.17</td>
<td>45.64%</td>
</tr>
<tr>
<td>512</td>
<td>269.96</td>
<td>369.41</td>
<td>45.14%</td>
</tr>
<tr>
<td>1024</td>
<td>303.06</td>
<td>459.79</td>
<td>43.36%</td>
</tr>
<tr>
<td>1024</td>
<td>276.2</td>
<td>295.92</td>
<td>44.44%</td>
</tr>
<tr>
<td>1024</td>
<td>316.97</td>
<td>389.88</td>
<td>43.16%</td>
</tr>
</tbody>
</table>

While PoLiMEr achieves good results when 58% of the power budget is assigned to simulation initially, the performance gains become more ambivalent when the initial power allocation is even more skewed towards the simulation. Figure 5.6 shows runtime differences between PoLiMEr and the baseline with 75% of the power budget assigned to simulation initially.
Figure 5.6: Runtime of LAMMPS with PoLiMEr’s power allocation against baseline. Simulation initially assigned 3/4 total nodes at 120 W, and analysis 1/4 of nodes at 120 W per node.

For reference, we include one run on 1024 nodes that exhibits slight performance degradation shown in Figure 5.7.

The same run is represented in Figure 5.8 which shows that PoLiMEr does not degrade performance uniformly. This figure shows the difference in the time it took for analysis nodes using PoLiMEr to reach every 20 synchronization steps and for analysis nodes using the baseline to reach the same synchronization steps. Similarly, the difference for simulation nodes is shown as well. Up to the 260th synchronization step, PoLiMEr performs better than the baseline. This synchronization step corresponds to the 178 second mark. Past this point PoLiMEr increasingly slows down the simulation, which leads to overall performance loss. In addition, after that point the change in power allocated to simulation and analysis between
steps stagnates. In other words, while PoLiMEr is actively shifting power from one to the other, there are marginal performance improvements, but once the power allocation remains roughly constant with subsequent synchronizations, no performance gains are achieved. This limitation is discussed in Chapter 6. Table 5.5 contains a detailed overview of the results in the case where simulation starts with 75% of the global power budget.
Figure 5.8: Difference between PoLiMEr and baseline in time to reach each synchronization point for one of the 1024-node runs. Before the 260th synchronization step PoLiMEr reaches synchronizations sooner than the baseline.

Table 5.5: Overview of results when simulation has more nodes and power initially.

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>PoLiMEr Time (s)</th>
<th>Baseline Time (s)</th>
<th>Final PoLiMEr % C' to simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>701.62</td>
<td>828.61</td>
<td>47.78%</td>
</tr>
<tr>
<td>64</td>
<td>686.63</td>
<td>826.12</td>
<td>47.25%</td>
</tr>
<tr>
<td>64</td>
<td>709.61</td>
<td>840.61</td>
<td>47.75%</td>
</tr>
<tr>
<td>128</td>
<td>479.14</td>
<td>527.14</td>
<td>47.65%</td>
</tr>
<tr>
<td>128</td>
<td>414.62</td>
<td>507.14</td>
<td>47.16%</td>
</tr>
<tr>
<td>128</td>
<td>469.65</td>
<td>556.61</td>
<td>47.64%</td>
</tr>
<tr>
<td>512</td>
<td>259.48</td>
<td>238.72</td>
<td>47.20%</td>
</tr>
<tr>
<td>512</td>
<td>213.17</td>
<td>209.34</td>
<td>47.04%</td>
</tr>
<tr>
<td>512</td>
<td>212.43</td>
<td>205.33</td>
<td>46.97%</td>
</tr>
<tr>
<td>1024</td>
<td>254.56</td>
<td>246.53</td>
<td>47.12%</td>
</tr>
<tr>
<td>1024</td>
<td>254.52</td>
<td>244.39</td>
<td>46.82%</td>
</tr>
<tr>
<td>1024</td>
<td>259.72</td>
<td>247.19</td>
<td>46.75%</td>
</tr>
</tbody>
</table>
5.2.3 Initial power distribution: simulation and analysis have equal power

Here we assign equal power and nodes to simulation and analysis. Figure 5.9 shows a summary of the runtimes achieved by PoLiMEr compared to the baseline. As the number of nodes scales, PoLiMEr cannot match the baseline. As was the case when simulation was assigned 75% of the power budget initially, PoLiMEr’s power allocation strategy matches and exceeds the baseline up to a certain point. Figure 5.10 shows the difference in time to reach synchronization between PoLiMEr and the baseline for one of the 1024-node runs. This time PoLiMEr outperforms the baseline for a shorter period. As mentioned before, the cause for this behavior is discussed in Chapter 6. Table 5.6 contains a detailed overview of the results.

Figure 5.9: Runtime of LAMMPS with PoLiMEr’s power allocation against baseline. Simulation and analysis assigned equal power and nodes.
Figure 5.10: Difference between PoLiMEr and baseline in time to reach each synchronization point for one of the 1024-node runs when simulation and analysis start with equal power.

Table 5.6: Overview of results when simulation and analysis have equal power and nodes.

<table>
<thead>
<tr>
<th>#Nodes</th>
<th>PoLiMEr Time (s)</th>
<th>Baseline Time (s)</th>
<th>Final PoLiMEr %C to simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>843.62</td>
<td>912.62</td>
<td>51.87%</td>
</tr>
<tr>
<td>64</td>
<td>888.61</td>
<td>902.11</td>
<td>50.43%</td>
</tr>
<tr>
<td>64</td>
<td>878.13</td>
<td>918.14</td>
<td>50.65%</td>
</tr>
<tr>
<td>128</td>
<td>450.19</td>
<td>469.19</td>
<td>51.16%</td>
</tr>
<tr>
<td>128</td>
<td>494.65</td>
<td>480.15</td>
<td>48.93%</td>
</tr>
<tr>
<td>128</td>
<td>632.15</td>
<td>685.76</td>
<td>51.78%</td>
</tr>
<tr>
<td>512</td>
<td>282.19</td>
<td>252.26</td>
<td>45.13%</td>
</tr>
<tr>
<td>512</td>
<td>374.80</td>
<td>225.84</td>
<td>45.38%</td>
</tr>
<tr>
<td>512</td>
<td>274.16</td>
<td>224.66</td>
<td>45.62%</td>
</tr>
<tr>
<td>1024</td>
<td>305.36</td>
<td>222.45</td>
<td>43.59%</td>
</tr>
</tbody>
</table>
CHAPTER 6
FUTURE WORK AND CONCLUSION

Several improvements can be made to make PoLiMEr’s power allocation mechanism more robust. Such improvements will address the main limitation seen in our results. The following is a list of considerations and improvements we will include in our future work.

• Special case: determining if power distribution is already optimal

In Section 5.2.2 and Section 5.2.3 we have seen performance issues at scale, where after a certain point at runtime PoLiMEr does not beat the baseline. The regions in which we see performance issues correspond to times when simulation and analysis are already close in time to reach a synchronization point, though not exactly. Although small (less than 1 second), the difference in time to reach the synchronization points is enough for PoLiMEr to attempt to reallocate power at times when it should not have. PoLiMEr makes this decision because there is a difference in time, and if the power distribution were already at an optimal ratio the times would have been equal. This raises the question of when is a difference in the observed times to reach a synchronization point due to system noise (e.g., node hardware variability), and when due to actual application properties? Since PoLiMEr actively collects application and system feedback during runtime, and since it keeps records of power and performance between synchronization points, changes can be made to our model to leverage this information and detect if power is already optimally distributed and to ignore noise.

• Special case: when more power does not improve performance

For some workloads other than the ones tested in this thesis, gains in performance with increased power could stagnate more quickly than for other workloads. This could yield to a situation where a workload’s performance would not change if PoLiMEr were to give it more power. At the same time, the second workload would be more power-constrained and slowed down, which would result in unused power. This power could
either be freed up for another job, or the second workload could be further slowed down
to conserve energy. Alternatively, power could be shifted back to the second workload.
This special case is not handled in our model yet.

- **Fitting the underlying power and performance function**

  Currently we assume a linear relationship between power and performance, which is a
good representative of many workloads, and has led to significant results in the majority
of the cases we evaluated. The true relationship between power and performance could,
however, be modeled more accurately based on past measurements and application
feedback that PoLiMEr maintains. We would expect to see even better results, and it
may alleviate some of our current limitations.

- **Power optimization opportunities within compute phases**

  As simulation and analysis don’t necessarily have the same amount of work, presence
of slack at a synchronization point is expected and we address this issue in this work.
Within the concurrent and independent compute phases there may be further power
optimization opportunities among the separate node partitions running simulation and
analysis.

- **Handling workloads that don’t operate at the power cap**

  As mentioned in Section 4.3, we expect many workloads to be running at or close to
the power cap in future power-constrained systems, but this may not always be the
case. The extension to handle this case is straightforward: we can limit power on the
low-power workload to its current power consumption and immediately shift the rest
to the other workload.

- **Evaluating more applications and in-situ visualization frameworks**

  We evaluated a widely-used and well-understood classical molecular dynamics simula-
tion with a set of analysis routines commonly used in this domain. In-situ interactive
visualization is another common form of analysis in HPC, and would serve as our next
evaluation target. Interactive visualization would introduce new challenges, such as more dynamic workloads due to user interactions.

- Simulation and analysis scheduled on same resources

Theta’s nodes contain a single power domain per node, which allowed us to formulate a solution for power management between simulation and analysis running on separate nodes. In future systems, there may be multiple power domains per node allowing us to consider cases where simulation and analysis execute on shared per-node resources.

6.1 Conclusion

Application-level power monitoring and management is necessary to efficiently couple scientific computing simulation and analysis workloads in power-constrained HPC systems. Typical in-situ analysis frameworks involve simulation and analysis workloads that periodically synchronize, which lends itself to power optimization opportunities that can be exploited with enough information from the application. To monitor, limit and actuate power at the application level, we propose PoLiMEr and use it to allocate power between simulation and analysis.

PoLiMEr can be used to gain valuable, fine-grained insight into power consumption characteristics of even complex simulation codes, node variabilities at scale, and application and system response to power caps, all of which point to potential resource management strategies. Key architectural features of the XC40 such as its memory hierarchy, shows new power saving opportunities even in the face of power constraints. We have shown how we can use PoLiMEr to identify DDR memory-intensive application tasks and apply a stringent power cap without any performance losses. Large-scale HPC applications with significant in-memory data requirements, such as data-driven science, are excellent candidates for future power capping studies. With its user-friendly API and abstraction of system-dependent power interfaces, PoLiMEr now enables HPC users of all backgrounds to engage in these
new research endeavors.

We extended PoLiMEr to allocate power between simulation and analysis in the presence of unused power and do so based on application feedback. We have shown that we can achieve significant performance improvements of a real in-situ analysis framework at scale with only a few pointers to PoLiMEr that mark synchronization phases. This suggests that the extensive domain knowledge of scientific application developers should be better leveraged and integrated with power management and optimization techniques for the next generation of HPC systems.
BIBLIOGRAPHY


First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization. ACM. 2015, pp. 19–24.


