THE UNIVERSITY OF WARWICK

Library Declaration and Deposit Agreement

1. STUDENT DETAILS

Please complete the following:

Full name: ……………………Oliver Francis John Perks……………………

University ID number: ……………………0501148……………………

2. THESIS DEPOSIT

2.1 I understand that under my registration at the University, I am required to deposit my thesis with the University in BOTH hard copy and in digital format. The digital version should normally be saved as a single pdf file.

2.2 The hard copy will be housed in the University Library. The digital version will be deposited in the University's Institutional Repository (WRAP). Unless otherwise indicated (see 2.3 below) this will be made openly accessible on the Internet and will be supplied to the British Library to be made available online via its Electronic Theses Online Service (EThOS) service.
[At present, theses submitted for a Master's degree by Research (MA, MSc, LLM, MS or MMedSci) are not being deposited in WRAP and not being made available via EthOS. This may change in future.]

2.3 In exceptional circumstances, the Chair of the Board of Graduate Studies may grant permission for an embargo to be placed on public access to the hard copy thesis for a limited period. It is also possible to apply separately for an embargo on the digital version. (Further information is available in the *Guide to Examinations for Higher Degrees by Research*.)

2.4 *If you are depositing a thesis for a Master's degree by Research, please complete section (a) below. For all other research degrees, please complete both sections (a) and (b) below:*

(a) Hard Copy

I hereby deposit a hard copy of my thesis in the University Library to be made publicly available to readers (please delete as appropriate) EITHER immediately ~~OR after an embargo period~~ of ………................... ~~months/years as agreed by~~ the ~~Chair of the Board of Graduate Stu~~dies.

I agree that my thesis may be photocopied. YES ~~/ NO~~ (*Please delete as appropriate*)

(b) Digital Copy

I hereby deposit a digital copy of my thesis to be held in WRAP and made available via EThOS.

Please choose one of the following options:

EITHER My thesis can be made publicly available online. YES / ~~NO~~ (*Please delete as appropriate*)

OR My thesis can be made publicly available only after…..[date] (Please give date)
YES / NO (*Please delete as appropriate*)

OR My full thesis cannot be made publicly available online but I am submitting a separately identified additional, abridged version that can be made available online.
YES / NO (*Please delete as appropriate*)

OR My thesis cannot be made publicly available online. YES / NO (*Please delete as appropriate*)

Addressing Parallel Application Memory Consumption

by

Oliver Francis John Perks

A thesis submitted to The University of Warwick

in partial fulfilment of the requirements

for admission to the degree of

Doctor of Philosophy

Department of Computer Science

The University of Warwick

July 2013

Abstract

Recent trends in computer architecture are furthering the gap between CPU capabilities and those of the memory system. The rise of multi-core processors is having a dramatic effect on memory interactions, not just with respect to performance but crucially to capacity. The slow growth of DRAM capacity, coupled with configuration limitations, is driving up the cost of memory systems as a proportion of total HPC platform cost.

As a result, scientific institutions are increasingly interested in application memory consumption, and in justifying the cost associated with maintaining high memory-per-core ratios. By studying the scaling behaviour of applications, both in terms of runtime and memory consumption, we are able to demonstrate a decrease in workload efficiency in low memory environments, resulting from poor memory scalability.

Current tools are lacking in performance and analytical capabilities motivating the development of a new suite of tools for capturing and analysing memory consumption in large scale parallel applications.

By observing and analysing memory allocations we are able to record not only how much but more crucially where and when an application uses its memory. We use use this analysis to look at some of the key principles in application scaling such as processor decomposition, parallelisation models and runtime libraries, and their associated effects on memory consumption. We demonstrate how the data storage model of OpenMPI implementations inherently prevents scaling due to memory requirements, and investigate the benefits of different solutions.

Finally, we show how by analysing information gathered during application execution we can automatically generate models to predict application memory consumption, at different scale and runtime configurations. In addition we pre-

dict, using these models, how implementation changes could affect the memory consumption of an industry strength benchmark.

Acknowledgements

This thesis, and the supporting research, was made possible by a great number of people. Their support throughout, both academically and personally, has maintained my focus and drive to research.

A special thank you to my supervisor Prof. Stephen Jarvis, for accepting me onto the PhD program, and supporting my research throughout.

After almost eight years at the University of Warwick, there are far too many people to thank individually, but certain people deserve special recognition. Simon Hammond, for helping me through the difficult first years of research and still being there until the end. All those with whom I work with on a daily basis: David Beckingsale, Bob Bird, Dr. Adam Chester, James Davis, Dr. Henry Franks, Dr. Matthew Leeke, Andrew Mallinson, Dr. John Pennycook and Steven Wright.

To Dr. Mark Pharaoh and Anna Wordsworth, my triathlon coaches, for keeping me sane and providing pastoral care throughout my PhD.

Andrew Herdman, Wayne Gaudin and others at AWE for providing the industry focus to my research, and general support.

Todd Gamblin for continued support of my research, and access to countless resources, both physical and intellectual.

Lastly to my family and loving girlfriend, Claire.

Declarations

This thesis is submitted to the University of Warwick in support of the author's application for the degree of Doctor of Philosophy. It has been composed by the author and has not been submitted in any previous application for any degree.

The work presented (including data generated and data analysis) was carried out by the author except in the cases outlined below:

- Initial collection of runtime and memory consumption results from the Lawrence Livermore National Laboratories was performed by Simon Hammond (Sandia National Laboratories).

Parts of this thesis have been previously published by the author:

- O. F. J. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. Should We Worry About Memory Loss?, SIGMETRICS Performance Evaluation Review, March 2011 [105]

- O. F. J. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. WM-Tools - Assessing Parallel Application Memory Utilisation at Scale, In *Proceedings of the European Performance Engineering Workshop* (EPEW), Borrowdale, UK, October 2011 [106]

- O. F. J. Perks, D. A. Beckingsale, S. D. Hammond, I. Miller, J. A. Herdman, A. Vadgama, A. H. Bhalerao, L. He, and S. A. Jarvis. Towards Automated Memory Model Generation via Event Tracing, The Computer Journal 56(2), June 2012 [103]

- O. F. J. Perks, R. F. Bird, D. A. Beckingsale, and S. A. Jarvis. Exploiting Spatiotemporal Locality for Fast Call Stack Traversal, In *Workshop on High-performance Infrastructure for Scalable Tools* (WHIST), June 2012 [104]

- O. F. J. Perks, D. A. Beckingsale, and S. A. Jarvis. Analysing the Influence of InfiniBand Choice on OpenMPI Memory Consumption, In *International Workshop on High Performance Interconnection Networks* (HPIN), July 2013 [102]

Sponsorship and Grants

The research presented in this thesis was made possible by the support of the following benefactors and sources:

- The University of Warwick, United Kingdom:
  Warwick Postgraduate Research Scholarship (2009–2013)

- Royal Society:
  Industry Fellowship Scheme (IF090020/AM)

- UK Atomic Weapons Establishment:
  "The Production of Predictive Models for Future Computing Requirements" (CDK0660)
  "AWE Technical Outreach Programme" (CDK0724)
  "TSB Knowledge Transfer Partnership" (KTP006740)

Abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
DDR	Double Data Rate
DBI	Dynamic Binary Instrumentation
DIMM	Dual In-line Memory Module
DRAM	Dynamic Random Access Memory
ECC	Error Correcting Code (Memory)
EDR	Enhanced Data Rate (InfiniBand)
FDR	Fourteen Data Rate (InfiniBand)
FLOP/s	Floating-Point Operations per Second
HCA	Host Channel Adapter (InfiniBand)
HPC	High-Performance Computing
HWM	High Water Mark
IB	InfiniBand
ISV	Independent Software Vendor
LANL	Los Alamos National Laboratory
LLC	Last Level Cache
LLNL	Lawrence Livermore National Laboratory
MHD	Magnetohydrodynamics
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
QDR	Quad Data Rate (InfiniBand)
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
SDRAM	Synchronous Dynamic Random Access Memory
SNL	Sandia National Laboratories

TDP	Total Power Draw

Definitions

Call Stack

The chain of function calls which brought execution to the current point.

Collective

Defines a type of communication with more than two parties involved, with multiple sources, multiple destinations, or both. Within MPI this represents function calls such as 'MPI_Gather' and 'MPI_Allreduce'.

Compute Bound

A computational operation whose time is primarily decided by the time taken to operate on the data, rather than the time to load the data into memory. In such a scenario the use of a faster processor will afford a proportional gain in overall performance.

Data Occupancy

The proportion of required data that is accessible in cache. High occupancy usually equates to high efficiency computation, and relates to a compute bound operation.

ECC RAM

Error-Correcting Code (ECC) RAM is a type of memory with built in error detection and correction, through the use of parity bits. This makes the memory immune to single bit errors, increasing reliability.

Ghost Cells

With parallel grid based computations it is frequently necessary to access data which resides in another processor's memory space. Such a situation usually occurs at the boundary of a processor's computational region. To improve the performance of fetching the data, a buffer is used to replicate the whole boundary region on the local processor. This data is rarely computed, but used as input to the computation of other cells.

High-Water-Mark

An application high-water-mark (HWM) is a measure of the maximum amount of memory consumed by the application at any point during its execution. In the case of parallel applications, we refer to HWM as being the maximum HWM of all of the processes within the job.

Memory Bound

A computational operation whose time is primarily decided by the time taken to load the required data into memory, rather than the actual computational operation. In such a scenario the use of a faster processor will not afford a gain in overall performance. Improvements in performance will only be afforded be improvements to the memory system, such as faster RAM or more memory bandwidth.

Paging

The process of swapping memory pages to disk, when system memory becomes full, thus increasing memory capacity. This introduces increased access overheads when these pages are swapped back into memory, and is traditionally not used in HPC configurations.

Parallel Efficiency

Is a measure of application scalability. It is defined by the parallel speedup $S(p)$, the parallel time $T(p)$ over the serial runtime $T(1)$, divided by the number of processors used p.

$$S(p) = \frac{T(p)}{T(1)} \tag{1}$$

$$E(p) = \frac{S(p)}{p} \tag{2}$$

Point-to-Point

Defines a type of communication with a single source and destination. Within MPI this represents 'MPI_Send' and 'MPI_Recv' function calls, and their associated variations.

Strong Scaling

The act of increasing the core count used to solve a problem of the same size.

Weak Scaling

The act of increasing the core count used to solve a problem of increasing size, where the size per core remains the same.

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

High performance computing (HPC) has become the cornerstone of many scientific disciplines, supporting experimentation through simulation to reduce cost and mitigate risk.

Whilst the benefits of HPC have been apparent to large scientific institutions for many decades, such systems have traditionally been prohibitively expensive for smaller companies and research groups. The rise of commodity computing has since lowered the machine cost and released the potential of HPC in many new domains.

Recent developments in parallel processing hardware, such as multi-core commodity processors, have had a dramatic effect on the performance of these supercomputing platforms. The evolving technologies constantly striving for increased performance at the high end of supercomputing result in a ‘trickling down’ of technologies, and their associated performance, to commodity computing. From historical analysis we can see that it only takes between six and eight years for the slowest machine in the Top500 list [82] to achieve the same computational power of the number one machine. Beyond the HPC domain we see that it only takes eight to ten years for a notebook computer to replicate the performance of that slowest machine in the Top500 [83].

Whilst the majority of media, and industry, attention has traditionally focused on the computational power of CPU architectures, the increased scale of modern HPC platforms has identified a number of other performance bottlenecks. Most scientific applications can be classified by their dependence on the four key components of computing: computational power; memory access latency and bandwidth; interprocessor communications; and I/O. The veritable neglect of these other components has stifled performance gains for many large scale scientific applications. To address this architectural imbalance the proportional component cost of supercomputers has been slowly evolving, and as such memory is consuming a larger portion of machine procurement budget than ever before.

In this thesis we focus on the role of computer memory in the HPC ecosystem, specifically focusing on memory capacity and utilisation. Whilst not directly a performance modifier, memory consumption and associated scaling will have a dramatic effect on available runtime configurations, dictating core counts and problem sizes. We also address how, in many situations, there is an inherent tradeoff between performance and memory usage, motivating the use of non-optimal techniques to reduce memory footprints and thereby enable the execution of larger problems.

1.1 Motivation

The rise of multi-core processors, and their increasing core count densities, is having a dramatic effect on memory-per-core ratios. The rate of technological development in computer memory (DRAM) has not matched equivalent advances in CPU architecture. Specifically, the rate of growth of DRAM capacity has not been proportional to the increase in CPU core counts. Where high capacity DRAM modules are available they are often prohibitively expensive, or their usage comes with performance limiting caveats.

Within HPC a job's memory is provisioned on a per-core or per-node basis, defining constraints based on physical resources and resource sharing policies. Failure to adhere to these restrictions results in job termination, often without warning or accurate error reporting, thus job configuration and accurate resource provisioning are crucial for maintaining a productive computational environment.

Predicting application memory usage is largely a 'dark art', comprised of an algorithmic expectation of usage and experimental experience of failing jobs. Historically it has often been cost effective to over-provision memory resources to prevent job failures, but with rising costs this option is becoming harder and harder to justify. As such it is increasingly important to accurately provision memory capacity resources during machine procurement, where the balance between cost and usability is most critical.

The rising scale of the jobs mean that it is harder to algorithmically predict memory usage, as other factors begin to play a more crucial role in memory consumption, and the failure of jobs is less acceptable as they waste valuable machine time. For this reason it is now key to fully understand the memory usage characteristics of key scientific applications in HPC institutions. Code users and system designers are increasingly looking towards software tools to provide and analyse this information.

Where application memory requirements are prohibitively disproportionate to others in an institution's workload, code engineers must look to reduce their memory consumption. This reduction can come in two different forms; either more memory efficient runtime configurations or the re-engineering of applications libraries. Again scientists look to tools to provide analytical insight into the cause of consumption, and validation of the results of any memory reduction exercise.

As such the tool chain supporting memory consumption analysis must be strong enough to support high level application analysis as well as fine gained analysis. We do not feel that these requirements are accurately provisioned for by existing memory analysis tools, and therefore propose the development of a new suite of tools to provide this level of data analysis.

1.2 Domain

Within this thesis we are concerned with the memory capacity, and associated resource utilisation, of parallel applications within the HPC domain. Whilst many of the topics we discuss have a broader scope, we limit ourselves to focusing on the application of such topics to supercomputers. In the case of our discussions on computer hardware we touch upon only a subset of technologies, to facilitate a more in-depth evaluation of HPC-centric hardware. Whilst elements of other hardware technologies may be applicable we try to maintain focus on current, and anticipated, industry trends.

Similarly the tools and methodology we present are not necessarily confined to HPC applications and architectures but we deliberately constrain ourselves to the field to demonstrate their capabilities.

1.3 Benchmark Applications

Throughout this thesis we make use of a wide range of benchmark applications. Each of these applications is used specifically to demonstrate the capabilities of different memory consumption analysis techniques. The choice of each application is designed to expose certain memory behavioural characteristics which are best investigated with the current technique. Although most of the techniques could be applied to all of the applications discussed, we do not believe that an interesting insight could be gained with every combination.

A full description of the applications used is presented in Appendix A.2.

1.4 Research Methodology

In this thesis we ask – to what extent can non-intrusive profiling methods be used to analyse application memory consumption? We look to provide code developers with the tools and methodologies to evaluate the precise reason for behavioural characteristics in memory consumption, with the ultimate goal of reducing application memory high water mark. Memory is an increasingly crucial component of parallel applications, with financial and physical constraints limiting future capacity. Such limitations are coupled with an evolving view of how parallelism is achieved, and how resources can be best utilised. Existing tools are very limited in their capabilities to deliver fine-grained analysis into how and where application memory consumption especially in relation to job scale. In this research we explore the capabilities of memory consumption analysis for increasing the scalability of applications, and in the process future-proofing them against trends in declining memory-per-core ratios.

To answer this question we employ quantitive methodologies. Through empirical analysis of parallel application we study relationships in memory consumption.

The early portion of this research is dedicated to the development of tools, of sufficient quality, to provide observational information regarding application memory behaviour. The latter is dedicated to the analysis of this information and the construction of hypotheses. Where possible we verify these hypotheses and evaluate their impact through more empirical testing.

1.5 Thesis Contributions

This thesis is based around the development of a memory consumption analysis tool, and its application in understanding different aspects of the memory utilisation of HPC applications.

Our analysis serves two purposes: the first is the understanding of an application's memory profile with sufficient detail to accurately provision hardware resources, either during job submission or machine procurement; the second is providing a much deeper level of understanding to facilitate code re-engineering to reduce memory consumption.

In addition to the presentation of our tools and methodologies we present three investigations into application behaviour, driven by a need to understand memory consumption behaviour.

Tools and Methodology We describe a methodology for collecting the memory allocation data of a parallel application, mid execution, and analysing the resulting data. We develop a suite of tools to achieve this data collection, and the associated analysis, to demonstrate the validity and capability of the methodology. For each method of analysis we present an example of where the tools can be used to improve understanding of the underlying application.

Scaling Analysis We show how memory consumption analysis can be applied to a full machine workload. We show how strong scaling can be used to facilitate the deployment of low memory machines, and analyse the viability of such platforms through efficiency analysis on simulated workloads. To further the potential of this method, we address some of the concepts and techniques for reducing ghost cells to improve memory consumption scalability without detriment to performance.

MPI Memory Analysis We apply our analysis to evaluate the impact of MPI implementation choice, for specific network hardware. By looking at the memory consumption attributed to MPI on different platforms, for different implementations, we identify the best configurations to improve memory consumption scalability.

Automated Memory Modelling We investigate the potential of using memory consumption traces to automatically generate predictive models for memory high water mark. Using allocation size comparisons we can predict memory consumption at different scales, both in terms of problem size and core count, with a high degree of accuracy based on data from only two runs. We further show how these models can be used to predict the memory effects of implementing new programming principles, such as 3D processor decompositions and hybrid MPI and OpenMP parallelism.

These contributions show how far non-intrusive methods of memory consumption profiling can be extended to provide critical analysis for application developers. We show how the collected data can be used to understand application consumption, and critically monitor the artefacts of consumption as the size of the job is scaled. The fine-grained analysis allows us to study potential issues at small scale before they become dominant problems at large scale. Additionally we show how this data can be used for predicting memory consumption at scales greater than those obtainable with current configurations.

1.6 Thesis Overview

The remainder of this thesis is organised in the following way:

Chapter 2 presents an introduction to computer memory, divided into two logical sections to discuss memory in terms of both hardware and software. We discuss why architectural nuances influence trends, and help determine memory capacity, coupled with an analysis of core counts to understand memory-per-core ratios. Additionally we discuss how memory is used within the system, and use this to understand differences in the way memory usage is reported to the user.

In Chapter 3 we continue with our introduction, but within the domain of software tools, presenting some key concepts and methodologies. We discuss the fundamental differences between tool types, and the roles that they play in the software ecosystem. Further more we discuss the methods of data collection utilised by these tools and evaluate their inherent performance properties. Within the domain of memory analysis, we present a discussion of a number of different software tools, evaluating them on the data they can collect and analysis they can perform. We use this evaluation to motivate the development of a new memory analysis tool, by identifying their limitations and desired functionality.

Chapter 4 presents the development of our memory analysis tool suite, WMTools, and features a demonstration of the various analysis methods applied to the AWE Chimaera benchmark application. We detail the key design principles, based on our previous evaluation of existing tools, and describe how our approach to data collection and, importantly, storage can be used for advanced analysis.

In Chapter 5 we investigate the memory scalability of applications, using strong scaling to reduce memory-per-core footprints. We use HWM data, with accompanying runtime data, to simulate the execution of mixed science workloads on computers with decreasing memory-per-core ratios. From this we are able to show the decrease in workload efficiency resulting from such usage, caused by the poor memory scalability of some applications. We further investigate the cause of poor memory scalability by demonstrating the impact of ghost cells, and present a number of memory reduction techniques such as processor decompositions and hybrid parallelism programming models.

Chapter 6 presents an investigation into MPI memory consumption, and the associated impact of communication buffers when strong scaling applications. Through analysis of a benchmark application we evaluate the memory scalability of different MPI implementations and different InfiniBand hardware. We experiment with a range of improved implementations to show how MPI memory consumption, which can grow at an alarmingly rate, can be reduced to more manageable levels.

In Chapter 7 we demonstrate the ability to generate automated memory models based on the execution traces from WMTools. We look at two cases: a simplistic model of Lare2D, where we construct two different models to capture the characteristics of two code regions; and the more complex modelling of Chimaera, where we capture the artefacts of ghost cells. We validate these models against results from strong scaling and a change in problem size. We then use the Chimaera model to generate predictions for memory savings based on 3D processor decompositions, and hybrid programming model, in accordance to the effects demonstrated in the latter half of Chapter 5.

In Chapter 8 we conclude this body of research by reiterating the contributions, discussing limitations to the research and detailing future work in the same domain.

Lastly in Appendix A we present a detailed evaluation of the architectures and applications used throughout this thesis.

CHAPTER 2
Background: Computer Memory

In this chapter we present an overview of the memory subsystem within modern computers, with respect to both hardware and software. We describe the different layers in the memory hierarchy and where our research interests sit. Additionally we look at current hardware trends, and from this identify a number of issues with current memory hardware. From this we discuss a number of future and emerging technologies, and describe the ways in which they address current problems in hardware.

From a software perspective we discuss the structure of applications in virtual memory, and how this relates to application memory consumption. We discuss different methodologies for measuring memory consumption and their limitations, and use this discussion to motivate our choice of measuring application level memory management in memory studies, both for accuracy and information availability.

2.1 Hardware Perspective

Figure 2.1 outlines the different layers of the memory subsystem hierarchy. The closer a layer is to the top of the pyramid, the closer it is to the logic units of the processor, and also the faster it is. The limitation of proximity is size, meaning to be close and fast the memory layer must also be small, thus the largest layers sit furthest away from the processor's logic units.

The majority of memory allocations are handled by the Random Access Memory (RAM), a layer built on the inherent tradeoff between capacity and performance. When the specific pages of memory are required by the application they are loaded into cache, a small but high-performance layer.

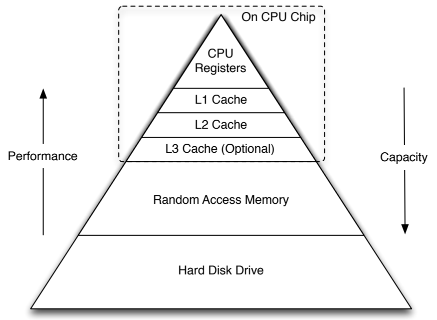


Figure 2.1: Hierarchal overview of traditional memory subsystem

In this thesis we are solely concerned with RAM, and the issues surrounding capacity limitations. The performance of RAM is already considered a bottleneck in memory bound applications, thus having to resort to another layer of the hierarchy to achieve sufficient capacity is not considered a viable option [34].

Within the HPC community the use of Error Correcting Code (ECC) RAM is standard, as a measure to reduce errors in calculations, thus increasing the accuracy of scientific applications and increasing overall system stability [42]. One of the downsides to ECC RAM is the increased cost, with chips often costing twice as much as equivalent non-ECC RAM. This is, partly, a result of requiring an additional storage for the parity bit, but mainly to do with the limited market as ECC RAM is not common in desktop computers.

Software level ECC, such as that provided in many Graphics Processing Units (GPUs), result in a capacity reduction of 12.5% with an associated reduction in memory bandwidth.

2.1.1 DDR Technology

Main computer memory is traditionally comprised of Double Data Rate (DDR) memory Dual In-line Memory Modules (DIMMs) composed of SDRAM memory modules. The current generation of DDR is DDR3, which provides a doubling of data rate at a reduced voltage over the previous DDR2 generation. The next generation, DDR4, is expected to be released in 2013 with support provided by architectures such as Intel's Haswell-E processors, with increased clock speeds at a further reduced voltage.

There are a number of competing DIMM technologies currently used within DDR3, each with a different emphasis on a different aspect of technology. Whilst these other technologies are available, RDIMM remains the most common DRAM used in HPC platforms due to the trade-off between cost, capacity and performance.

Ranks

DDR DIMMs are often referred to in terms of ranks: single, dual, quad and the emerging octal rank. The rank describes the number of modules, group of similar DRAM chips, on a DIMM. Each rank has a maximum capacity based on DRAM chip capacity and payload.

For example a quad rank RDIMM can support up to 32 GB, with a rank width of 72 bits for ECC support, by utilising 16 $\times$ 4 Gb DRAM chips each rank can support up to 8 GB [52].

Thus to achieve higher DIMM capacity one must either increase the number of ranks, or the the rank capacity. The introduction of 8 Gb DRAM technology will facilitate the development of 64 GB RDIMMs, alternatively moving to octal rank LRDIMMs which can support up to 64 GB [84].

RDIMM

Registered DIMMs contain a register to buffer the address and control lines in addition to the clock. This makes a more stable memory system, allowing for higher capacity DIMMs and, often, the use of more DIMMs per channel. This buffering does introduce a minor latency, and can reduce bandwidth when using a single DIMM per memory channel. The additional hardware and relatively low market prevalence result in a slight increase in cost over UDIMM.

UDIMM

Unregistered DIMMs are slightly cheaper than comparable RDIMMs, at smaller capacities (2 GB to 4 GB). Each UDIMM installed introduces electrical load and issues with noise [63]. Achieving higher capacities without buffering is harder, meaning there is limited availability of the highest capacity DIMMs (8 GB), and they are often more expensive than RDIMMs. Within HPC the use of UDIMMs is not popular, due to the unattractive price-per-GB and the restrictions on configurations.

FBDIMM

Whilst FBDIMMs are no longer favoured, their technological achievements make them an interesting point of discussion. Fully Buffered DIMMs were designed to increase the number of DIMM slots supported on each memory channel, by providing an area of on-chip memory to act as an advanced memory buffer (AMB) [43]. Through the use of a serial bus the memory channel would communicate with the AMB providing some error correction and facilitating the issuing of parallel read/write commands, as they can be buffered.

In addition to increased cost FBDIMMs were plagued with power usage and heat dissipation issues [74], and were not widely adopted. Many manufacturers have since ceased production, and removed the technology from their roadmap. The technological benefits behind FBDIMMs are very desirable and work to replicate them is still underway; we discuss some of these projects in Section 2.1.5.

LRDIMM and HCDIMM

Load Reduced DIMMs and Hyper Cloud DIMMs are very similar in concept, as an amalgamation of RDIMM and FBDIMM technology. In addition to the buffering done by RDIMMs they buffer the data lines, making all lines fully buffered. The reduction in electrical loading on the chips, due to the buffer logic, enables the processors to drive more DIMMs at a higher clock speed.

Quad rank LRDIMMs utilise their rank multiplication to present their four logical ranks as two virtual ranks, enabling three quad rank chips to appear as six virtual ranks. The increase in rank capability enables the use of higher capacity DIMMs.

HCDIMM differs from LRDIMM by providing logic support without the need for BIOS configuration. This is claimed to provide an increase in bandwidth and throughput [95].

The cost of LRDIMM, and HCDIMM, is slightly more than RDIMM due to the additional logic and limited market prevalence.

DDR Data Rate

DDR stands for double data rate. This is the process of ‘double-pumping’ data with regards to the clock – data is transferred on both the rise and the fall of the clock signal. This results in a slight confusion of terms, generally amongst vendors, between clock rate Megahertz (MHz) and data rate Mega-Transfers per second (MT/s). A chip labeled DDR3-1600 will have a data rate of 1600 MT/s, resulting from a clock speed of half that figure, 800 MHz, though many vendors, and other sources, may incorrectly label it as 1600 MHz.

During this thesis we refer exclusively to the data transfer rate to avoid confusion.

2.1.2 Trends in Memory Architecture

In this section we will look at some of the underlying trends in hardware technology and machine architecture. Using the ‘CPU DB’ data set from the Stanford VSLI group [28] we are able to track architectural changes in a number of key regions for a wide range of processors from the past 40 years. We note that whilst this data set is significant it does lack details of a number of more recent architectures. Additionally we compile data from the Top 500 list of supercomputers [82] looking at the historical trends of the largest supercomputers in the world. The third data set we utilise is Intel ARK [55], a product

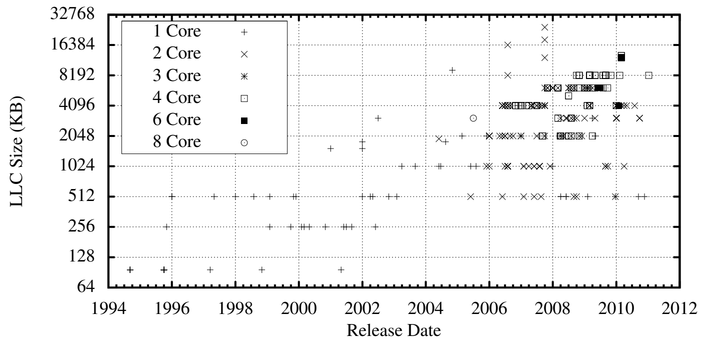


Figure 2.2: LLC size of processors over time

specification database. Specifically we use information collected about the server grade products: the E7, E5, E3, 7000, 6000, 5000 and 3000 product lines.

Kogge and Dysart presented a more comprehensive analysis of the Top 500 list, evaluating historical trends and making future projections based on roadmaps [65]. From this study they are able to observe the fall in the bytes / FLOP metric, a comparative measure of storage to compute capacity. They attribute this, in part, to the rise of lightweight and heterogeneous systems.

Cache Size

Figure 2.2 plots the size of the Last Level Cache (LLC) (either L2 or L3 depending on the architecture) against the processor's release date. Additionally we have grouped the data by the number of cores on the processor, which allows us to view the different trends in architecture design. We can clearly see that over time the LLC size is increasing, signifying its importance in modern day computing. There is also no discernible trend between core count and cache size; a number of single and dual core chips have equivalent cache sizes as chips with higher core counts. Some patterns are clearly visible though, like the preference for 2 MB per core, resulting in a number of quad core chips with 8 MB of LLC and dual core chips with 4 MB of LLC.

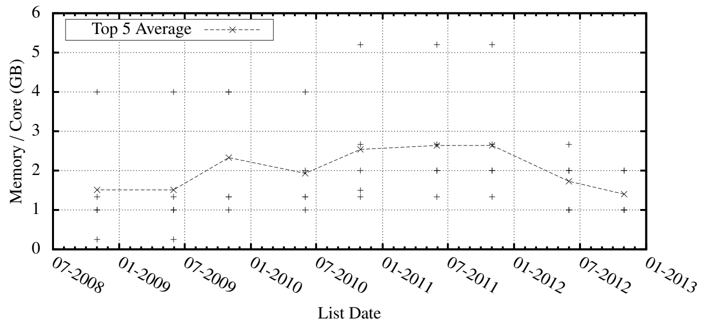


Figure 2.3: Top 5 supercomputer memory per core ratios

Memory Per Core

Using the Top500 list we collect information on the top five systems from each list, and plot their memory per core ratios. For heterogeneous systems we represent only the ratio of DRAM to CPU cores, and exclude GPU processing and memory capacity.

Figure 2.3 shows this trend over the last five years. What we see from this graph is that the average memory per core of the top five systems is actually quite constant, generally between 1.5 GB and 2.5 GB per core.

Whist we do not observe a decline in memory-per-core ratios, as expected, we attribute this to the extreme scale of the platforms analysed. At the top end of HPC price is always a dominating factor and so memory capacity has always been constrained. Although we do not have sufficient data to analyse the full 500 machines in the list, we would expect to observe some capacity-based trends. Whilst we observed in Chapter 1 that it only takes six to eight years for the performance of the top computers to be amortised into the lower echelons to the Top 500 list, we do not believe this trend to hold true for memory-per-core ratios.

2.1.3 Hardware Capacity and Costs

The memory capacity of computers is controlled by two factors: physical capability and financial constraints. In this section, we discuss how memory architecture influences memory capacity, through configuration options, and briefly investigate the financial considerations.

From this we show that the challenges to maintain existing memory-per-core ratios is managing the DIMM configurations to minimise financial overheads. In other, more memory constrained, domains the limitations on memory capacity are resulting in the proposal of novel architectures such as disaggregated memory systems [73].

Memory Channels

Current generations of Intel processors support up to four memory channels per processor [55], which determines bandwidth. Each memory channel can normally support three DIMMs [88, 89] of up to 32 GB in capacity. This gives a theoretical maximum capacity of 768 GB for a two socket, four memory channel, three DIMMs-per-channel (DPC) 32 GB DIMMs configuration, though such a configuration is highly expensive.

Intel does support up to four DPC through the use of the Intel 7500 generation chipset, which supports Scalable Memory Buffer (SMB) [56, 57]. Building on the principles of FBDIMMs, the SMB acts as an intermediary operation buffer, but utilises traditional DDR3 RDIMMs. Each SMB provides two memory channels, each supporting eight logical ranks, allowing the use of four quad rank chips per SMB.

The 7500 chipsets are generally used for very memory dense servers, theoretically allowing up to 4096 GB, when the 128 DIMM slots of an 8-socket server are populated with 32 GB DIMMs.

With regards to memory systems, Intel and AMD based architectures are very similar, and so assertions illustrated with Intel based examples are also generally applicable to AMD as well. The AMD ‘Magny Cours’ micro-architecture exhibits many of the same memory subsystem configurations as Intel’s ‘Nehalem’ micro-architecture including an integrated memory controller [23]. Both architectures support up to four memory channels per CPU, each supporting three DPC.

The AMD ‘Interlagos’ architecture is slightly different as each CPU is made up of two logical dies, with each die containing up to four ‘Bulldozer’ modules (two ‘cores’ with a shared floating point unit). Each die has its own memory controller with two memory channels, giving a total of four channels per CPU.

The IBM Power architecture is similar to the Interlargos architecture in the way that multiple dies, with independent memory controllers, are contained within the same chip [53].

The future Haswell-E architecture from Intel, which will support DDR4, is rumoured to contain only four memory channels, similar to current generations, and be limited to one DPC due to the use of octal rank DIMMs.

Hardware Limitations

Although the memory capacities discussed above are theoretically achievable, in practice it is not always that easy. Whilst the use of three DPC is supported configuring such a system is non trivial [11, 41, 120]. UDIMMs only support a maximum of two DPC, with a maximum combined capacity of 64 GB.

Memory channels can support a maximum of eight logical ranks each, this means up to two quad rank DIMMs or a mixture of three dual or single rank DIMMs. 32 GB RDIMMs are only available in quad rank, which is only supported in two DPC mode, thus limiting the maximum memory capacity to 64 GB per channel.

LRDIMMs can be run in three DPC mode with ‘quad rank’ chips, as they will appear as a combined six virtual ranks, enabling up to 96 GB per memory channel. One limitation is that such a configuration can only be supported in reduced performance mode, reducing the I/O clock speed from 666 MHz to 533 MHZ, resulting in a reduction of transfer rate from 1333 MT/s to 1066 MT/s.

Additionally some system configurations will not allow the use of low voltage DDR3 DIMMs (1.35 V as opposed to 1.5 V) when utilising three DPC, due to

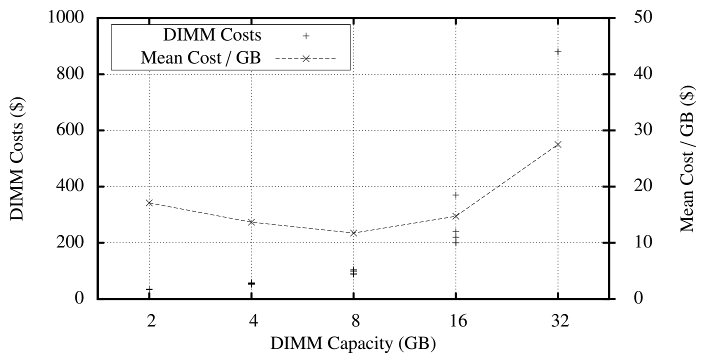


Figure 2.4: DRAM cost capacity comparison

the increased noise and signal degradation.

DIMM Cost

To maintain established memory-per-core ratios, within nodes with increased core counts, nodes must increase their memory capacity accordingly. Often it is not just the case of adding more memory DIMMs to the specific nodes.

In Figure 2.4 we present an evaluation on the current list price of DDR3 RDIMM chips with ECC support from Crucial as of June 2013 [25]. We present the cost-per-DIMM of varying DIMM capacities, factoring in the cost of different speeds and configurations, in addition to the average cost-per-GB of each capacity DIMM. What we see from this figure is that whilst initially the cost-per-GB decreases, to a ‘sweet spot’ at 8 GB DIMMs, the cost-per-GB then starts increasing, demonstrating that higher capacity DIMMs are less cost effective.

In a DIMM count-constrained environment the system architect will have to utilise higher capacity DIMMs to maintain memory-per-core ratios in high core count configurations. The implication of this figure is that such a configuration is less efficient in terms of cost-per-GB, thus driving up the total cost of the machine.

Whilst we note that these prices only reflect those available to public consumers, at small purchasing scale, we believe the trend to be representative of larger procurements. A discussion on relative DIMM costs by Fujitsu also concluded a price-per-GB sweet spot at 8 GB DIMMS [41]. They also highlight how UDIMM is comparatively cheaper than RDIMM, and LRDIMM is comparatively more expensive, supporting our descriptions in Section 2.1.1.

Power Consumption

Whilst the power consumption of individual DDR3 DIMMs is considered to be fairly low, the cost is multiplied by their sheer multitude. There are two factors to take into account with DIMM power consumption: idle power and active power.

HP report the active power consumption of an 8 GB dual rank low-voltage (1.35 V) RDIMM to range from between 3.5 to 5 W, depending on speed [50]. This falls to about 0.5 W when idle, regardless of speed.

If we take for example a cluster, similar in configuration to Bull's Kay platform, of a 1000 dual socket nodes utilising oct-core Intel E5-2580s, with a maximum Thermal Design Power (TDP) of 130 W and 8 GB low-voltage RDIMMs. Maintaining a memory-per-core ratio of 4 GB will consume in the region of 40 kW when actively used, compared to the ≈20 kW required to maintain an active 2 GB-per-core ratio. This is in contrast to the theoretical maximum of 260 kW consumed by the processors in such a system, but still represents a significant power consumption saving.

These figures also motivate our research into memory consumption reduction, as the potential power savings from reducing memory use are significant.

2.1.4 Challenges in Memory Technology

There are three challenges with current memory technology that need addressing in the future: capacity, performance (latency and bandwidth) and power. The technological requirements of these problems can be largely distinct, for example there is a natural trade off between bandwidth and power consumption, but progression in all three domains is required.

The Memory Wall

The original concept of 'The Memory Wall' addressed the issue of the performance gap between processors and memory with respect to latency [128]. The concept states that the discrepancy between the rate of improvement between processor performance and memory latency would eventually lead to a state where every operation is memory bound, that processing would be so quick that the defining limiter on computation time would be the memory fetches for each piece of data that needed computing.

Technological developments have drawn out the effects of 'the wall' but it is still anticipated to influence future platforms.

Rise of Many-core

One of the biggest changes in the performance landscape, from a memory perspective, is the rise of many-core. Multi-socket systems often utilise separate NUMA regions, enabling a duplication of memory subsystems. Many-core, on the other hand, increases core count but using the same fundamental memory subsystem.

The additional resource contention, resulting from more cores utilising a similar number of memory channels, has lead to a stark performance gap. The increase in processor cores has not been met with an equivalent quantity of increases in other regions of the memory subsystem [79].

One of the implications, of particular interest to this body of research, is the impact on memory-per-core ratios. Whilst DIMM capacity is increasing, albeit slowly, the only way of maintaining memory-per-core ratios is to increase DIMM count. This has implications for both power and performance, but the biggest implication is for cost, as large capacity DIMMs and server grade motherboards with high DIMM counts are expensive. On accelerator architectures there is no user configuration of memory capacity, users are restricted to buying hardware to match their memory requirements from the limited selection available. Thus maintaining memory-per-core ratios is even harder when accelerators are factored in.

Power Usage

One of the major challenges identified by a leading group of experts in the DARPA ExaScale Computing Study is energy consumption [14]. Not only did they perceive it to be a highly critical challenge but also one where there is a significant gap between current and required technology.

One of the technologies identified as problematic for the projected power requirements is DRAM, with issues raised over both capacity and utilisation. They do not predict a reduction in the memory capacity needs of future supercomputing systems, but do identify a need to reduce power consumption.

Whilst there is a move to increase DRAM power efficiency, there are fundamental issues which plague the technology. The use of different 'power states' in memory can be an efficient way to reduce power consumption, but the state transition time can have a marked performance impact [32]. DDR3 currently supports a 1.5 V power supply, with a low voltage option at 1.35 V; DDR4 will utilise a 1.2 V power supply [61] with a further low power mode predicted at 1.05 V [85], though at lower voltages signal degradation becomes a considerable factor. One alternative is to dynamically frequency scale the memory controller, enabling memory power savings with only a small performance cost [31].

2.1.5 Future Technologies

To address the current problems in memory architecture, we look to future technology. We can see how different technologies are addressing the current limitations in existing technology, and what the implications are for future systems. Emerging technologies must provide improvements in a number of the different problem domains for them to be viable in future systems. Where technologies address only a single problem, they must be compatible with other technologies which afford similar enhancements in different domains [19, 127]. The combination of such technologies is hoped to bridge the gap between the power and efficiency developments of CPU architectures and the current state in memory and interconnect technology [129].

Silicon Photonics

The idea of silicon photonics as viable optical interconnect fabrics for both on- and off-chip communication is becoming increasingly appealing [14]. The low power consumption and high bandwidth make it an appealing choice in both scenarios. Further, silicon photonics can be utilised to communicate between processors and DRAM, to enhance performance at reduced power levels [9, 10].

3D Stacked DRAM

3D memory is an emerging technology designed to increase performance and decrease power consumption [1, 68]. The fundamental idea is to stack existing memory technology in a 3D structure which maximises density and minimises internal distances. The use of Through Silicon Vias (TSVs) is predicted to be the most efficient means of connecting layers for the highest density [20, 27].

One of the biggest improvements from stacked DRAM is the potential proximity to the processor, allowing for significant increases in bandwidth [76, 77]. Simulations of 3D DRAM technologies have identified the performance gains to be significant, in the order of 20% with an associated reduction in energy consumption of $\sim$7% [18].

NVIDIA announced at their 2013 GPU technology conference their intention to include 3D stacked DRAM in the Volta GPU, scheduled for release in the next four years [2]. They estimate achieving a bandwidth of $\approx$1 TB/s, significantly faster than the current 250 GB/s of their Kepler K20X [96].

Hybrid Memory Cube

The Hybrid Memory Cube (HMC), developed by Micron, is an extension of the idea of 3D stacked DRAM, with the addition of a logic layer [60]. This format allows for high-density memory, thus providing an increase in both performance and power efficiency.

The current HMC implementation contains 1 GB of DDR, and while it is expected that capacity of such devices will increase, they are unlikely to compete with traditional DRAM on price-per-GB in the early days of availability. As such it is hard to predict the role HMCs will play in future architectures, as to if they will replace current DRAM, with a probable loss of memory-per-core, or be used as supplemental memory in another layer of the hierarchy, or as part of a hybrid system [126].

Processing-In-Memory

The idea of Processing-In-Memory (PIM) is not particularly new, as it was first discussed in 1970 [118], but is still an active area of research today [64, 100, 33, 130]. The fundamental idea is a tighter coupling of storage and computing, by integrating a number of vector units into, or very close to, DRAM. The increase in proximity between storage and compute enables a reduction of latency and increase of bandwidth for certain memory operations [47, 109].

The Data intensIV Architecture (DIVA) is a PIM-based system coupled with a conventional microprocessor. Draper *et al.* show, via an extended RSIM (Rice Simulator for Instruction-level Parallelism Mulitprocessors) simulator, to provide an average performance speedup of 3.3× across a broad spectrum of eight scientific applications [33]. The majority of the performance gain is attributed to decreases in the memory stall time, afforded by the PIM, with further performance gains from their 'WideWord' unit being able to further exploit fine grained parallelism.

Buffer-on-board Memory

The idea behind buffer-on-board memory is to insert an additional logic layer between the on-die memory controller and the DRAM DIMMs [21, 24]. This logic layer, in the form of a simple memory controller, is designed to handle requests from the memory controller, and return the data back to it. The benefit from this format comes from the properties of the intermediate bus connecting the two memory controllers, which is designed to be narrower and faster than a conventional memory bus.

The ambition is to replicate some of the qualities of FBDIMMs, without the negative impacts on latency or the power and heat dissipation issues. By buffering and re-sending data certain issues with signal quality degradation can be alleviated. Additionally, this enables parallel operations, again enhancing performance.

As a combination of the concepts of PIM and buffer-on-board memory, an Active Memory Controller (AMC) can perform certain scalar and stream operations on cache-coherent data from within the memory controller [36].

2.1.6 Accelerators

Whilst this thesis is focused on the analysis of main memory, it is worth discussing the role of accelerators, e.g the NVIDIA GPU range and Intel Xeon Phi co-processors, and their memory in HPC platforms. Not only are accelerators becoming increasingly popular in HPC, due to their high density of computational power, but they also present a very interesting perspective for memory.

At the time of writing, accelerators sit on the other end of a PCI bus, which is often considered a performance bottleneck due to the high cost of data transfer, motivating increases in code residency for performance [22].

When accelerators are utilised in ‘off-load’ mode, where the host CPU controls the device and sends data to and from main memory. Unless memory is allocated specifically on the device, both the host and the device must have sufficient memory for the computation. In ‘native’ mode, where the device acts independently of the host, only the device requires enough memory for the problem.

DRAM Technology

Accelerator memories are traditionally based on the Graphics Double Date Rate (GDDR) SDRAM memory technology, due to the increased bandwidth. The increased latency associated with the technology is hidden by the increased level of parallelism associated with the processing component in addition to an 8 bit prefetch scheme.

The latest NVIDIA Kepler GPU, the K20X, has six memory controllers, each driving 4×256 MB GDDR5 chips, giving a total of 6 GB across the 24 chips and providing a total memory bandwidth of 250 GB/s. When compared to one of the Intel Sandy Bridge generation processors, such as the 8-core E5-2680, which have four memory channels, there is a total bandwidth of 51.2 GB/s.

Memory-per-Core Ratios

Memory-per-core is a non-obvious metric for accelerators, due to the complexities of defining a comparable ‘core’ unit between accelerators and traditional CPUs. The NVIDIA K20X consists of 14 streaming multiprocessors (SMX) – a comparable unit to a conventional core, each comprised of 192 CUDA cores.

The card provides 6 GB of GDDR5 memory, of which 5.25 GB is accessible when ECC is enabled. This gives a memory ratio of 384 MB per SMX and 2 MB per CUDA core.

The first generation Intel Xeon Phi coprocessor (“Knights Corner”, 7120X) has 61 lightweight cores, each supporting 4 threads, totalling 244 threads with access to 16 GB of onboard GDDR5 memory. This gives a memory-per-core ratio of ≈269 MB per core, and ≈67 MB per thread.

Obviously these memory ratios are much lower than conventional memory-per-core ratios partly as a result of the programming models supported by these architectures. A fairer comparison would be that made at the socket level, where 16 GB is well within the normal range for a modern 8-core processor.

Whilst conventional processors can be considered latency-bound the high degree of parallelism in accelerators make them throughput-bound. As such bandwidth is a more important metric than memory-per-'core', and the volume of device memory only dictates the proportion of a problem which can be resident at any one time.

2.2 Software Perspective

In this section we evaluate the memory system from a software perspective, focusing on how applications interact with the operating system and in-turn the underlying hardware. From this we can then evaluate how memory allocations are handled and the different methods of measuring memory consumption.

2.2.1 Allocations

Different programming languages have different methods of allocating space, such as Fortran's 'allocatable' or C++'s 'new'. These calls all result in POSIX user level memory management calls, such as malloc and calloc. These allocations are, in turn, handled by the memory allocator which manages the virtual address space. The allocator will request memory pages from the operating system via one of the two system calls, 'mmap' and 'brk'. Although theses system calls are generally only used by the memory allocator they can be called explicitly by the user for advanced memory management.

Malloc Malloc is the standard memory allocation function, which returns a block of virtual memory address space of the given size (in bytes). The memory is not automatically initialised to a value or touched, and so the allocator may employ lazy allocation principles.

Calloc Unlike malloc, calloc initialises all of the allocated bits to zero thus preventing lazy allocation. Calloc is designed for allocating a block of memory composed of an array of elements of specified size.

Realloc Realloc is designed for extending a region of memory, or potentially shrinking it. Where contiguous space in the virtual address space is available the call extends the memory region in-place, otherwise it relocates it, in both cases the content is preserved.

Free The free function call returns a block of memory to the allocator, and enables the region to be used by other allocations. Unlike languages with garbage collection, in POSIX languages free must be called explicitly to deallocate memory.

mmap Memory Map (mmap) is an allocator for larger, independent, chunks of memory. This methodology reduces fragmentation, as chunks can be independently deallocated. In addition to mapping physical memory to virtual memory 'mmap' can be used to map other devices, or even files, to memory. We note that whilst this memory is not technically part of the virtual memory heap we do not differentiate between the two.

brk The use of 'brk' is a more traditional way of obtaining more memory from the operating system. It does this by expanding the current data area in a continuous block. A call to 'brk' with a negative value will shrink the data area, again in a continuous block. Calls move the 'program break' location, defining the end of the data segment for that process. Due to the use of continuous regions of memory, fragmentation can occur, which can prevent the whole data region from being shrunk. To minimise these effects of fragmentation, 'brk' is traditionally only used for small allocations.

The research in this thesis tends to sit at the application level, and thus is allocator agnostic. As such our experiments are based on the default system

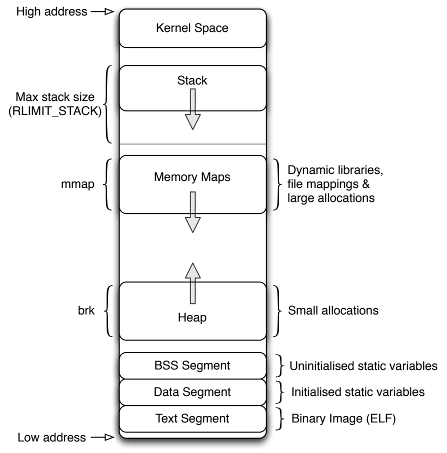

High address
Kernel Space
Max stack size (RLIMIT_STACK)
Stack
mmap
Memory Maps
Dynamic libraries, file mappings & large allocations
brk
Heap
Small allocations
BSS Segment
Uninitialised static variables
Data Segment
Initialised static variables
Text Segment
Binary Image (ELF)
Low address

Figure 2.5: Structure of virtual memory address space for application allocator.

Lazy Allocation

Many operating systems operate on the principle of lazy allocations. This means that not all of the memory allocated is mapped from virtual memory to physical memory instantly.

With a large allocation, malloc may return a valid virtual memory address, but not all of the pages are mapped to physical pages. Only when a page is accessed is it actually mapped. This behaviour allows the operating system to oversubscribe hardware pages, between applications, making use of space which has been allocated but not used. This can cause Out Of Memory (OOM) issues with the system, where the OOM killer will kill processes to free memory.

This behaviour is one of the main differences between malloc and calloc, as calloc initialises the memory, thus invoking the mapping. This ensures the memory is actually available when the application comes to use it.

2.2.2 Virtual Memory

Figure 2.5 represents the layout of a C style application in virtual memory on a Linux style operating system. From this it is clear to see the regions of memory where consumption can occur. In the figure we differentiate between the memory map region and the heap, and present more discussion on these regions in Section 2.2.1.

Memory, from an application perspective, is generally split into two locations: the stack and the heap (including the memory mapped region).

The Stack This region is generally used for small bits of data, such as variables and function parameters. It is composed of 'stack frames', which represent the state of the current function. Stack frames are stacked in order of call, thus are easily traversable and accessible through a single stack pointer.

Memory Mapping Segment This region of memory is consumed by calls to 'mmap', and is for use for dynamic libraries and the mapping files or devices to memory. Memory maps can also be used by allocators to store large objects.

The Heap This region is managed by calls to 'brk', which maintains a continuous data region. The majority of allocations are served by this data region. The term is used throughout this thesis to refer to the memory region consumed by allocations, though the allocator may use either 'brk' or 'mmap' to handle the specific allocations.

2.2.3 Reporting Consumption

The use of virtual memory management in Linux makes the recording of memory consumptions somewhat complex. It introduces subtleties of what should be viewed as the memory consumption of an application, and how it should be reported.

Virtual memory is the memory space visible to the application. It appears continuous, but is actually mapped to a number of pages of physical memory. Pages inside the virtual memory space may or may not actually be loaded into memory, depending on current usage. Similarly different pages of physical memory can be shared between multiple virtual memory spaces, in the form of shared memory.

We detail a few of these perspectives here, and discuss the role they play in the research presented in this thesis. Of these, one focuses on application level and the remaining three focus on the operating system level analysis.

Allocations

At the highest level the allocation's size is the volume of memory requested by the application. This is the memory consumption we choose to focus on, as it is closely coupled to the application. As such this volume should not differ between platforms or operating systems, though the memory consumed by system libraries may do.

We note that due to the specifics of allocator memory management, discussed in Section 2.2.1, we must measure allocations at the application level, as opposed to the system level. Monitoring the system calls to 'mmap' and 'brk' would only indicate the memory assigned to the allocator, rather than what the application specifically requested. As such there is no allowance for understanding the effects of lazy allocations here; we simply record what the application requests, and not how the operating system handles the allocations.

Virtual Memory Size

Virtual Memory Size (VSZ) represents an over-prediction of memory consumption, by recording a total of everything that is used and everything that can be used. This represents the complete size of the virtual memory address space. This size also includes memory which hasn't been loaded into physics memory pages. As such this metric is very close to measuring allocations at the application level, except with reduced information about the structure of the allocations.

Resident Set Size

Resident Set Size (RSS) represents a total memory consumption which includes shared memory. Whilst shared memory can be accessed by multiple different threads, the RSS value of all owning threads will include the full volume of the shared memory in their report.

Proportional Set Size

Proportional Set Size (PSS) is seen as a fair way of representing memory consumption, but accounting for a proportional cost of shared libraries. Rather than RSS assuming each process takes full account of the memory used by shared libraries PSS splits the consumption of each shared library, by the number of processes sharing it.

This method allows heavily shared libraries to contribute a small volume of consumption to each process utilising it. This is especially applicable within the domain of HPC, where multiple instances of the same application are loaded on the same node.

2.3 Memory Reduction Techniques

Understanding how an application utilises memory, and how virtual memory is translated onto hardware, lets us investigate potential methods of reducing memory consumption at the system level. In this section we explore a number of these memory reduction techniques and discuss how applicable they are to HPC environments.

2.3.1 Memory Deduplication

Memory deduplication is the process of consolidating replicated pages of memory to reduce storage requirements. It is a technique which can be applied to multiple domains such as filesystems [15, 132], virtual machines [123] and, of particular interest, HPC applications [71, 117].

The processing overheads of deduplication can reduce the appeal for many HPC applications, as the benefits are very application dependent.

2.3.2 In Memory Compression

One alternative to deduplication is compression, where data is stored in RAM in a compressed format to increase 'virtual' capacity [122]. One such example is the IBM Power7 system which has support for Active Memory Expansion (AME), where the operating system transparently manages both compressed and uncompressed pools of memory [48]. In memory constrained situations data can be compressed to free up space, then when required it can be decompressed. The compression ratios and associated cost are workflow- and data-dependent, so the AIX operating system comes equipped with a tool to evaluate workflows for the ideal compression factor.

2.3.3 AMR

Adaptive Mesh Refinement (AMR) is an algorithmic approach to memory reductions by limiting regions of interest in a uniform fine-grained mesh. The technique uses a globally coarse mesh whilst maintaining multiple levels of refinement on these areas of interest, down to the resolution of the uniform mesh.

The benefit of this method is the global reduction in computational complexity and memory consumption. Parallel implementations must use load balancing algorithms to evenly distribute patches, to ensure that each processor has a fair proportion of computational load.

Initial implementations of the AMR technique demonstrated a 5.2× reduction in memory requirements for the global mesh, for the given 2D problem [13]. A subsequent implementation of a 3D AMR technique saw a 22.1× reduction in memory requirements for the specific problem [12].

AMR is a very popular technique in HPC applications as it can enable the computation of larger problems, which would otherwise not fit in memory or would otherwise take too long to compute.

2.4 Summary

In this chapter we have detailed the different types of memory and their purpose in the memory hierarchy. We have demonstrated the past trends of the technology, identifying the root cause of memory capacity problems in modern systems.

From a software perspective we have explained the role of memory at both an application and system level. We have discussed the different types of allocations, and how the system interprets these calls. Building on this we have compared the different memory consumption metrics available, and motivated our design decisions.

By looking at consumption from an application level, by capturing memory management function calls, we can get an accurate representation of memory consumption. Whilst this does not allow us to interpret how the operating system has translated these requests for memory into hardware pages, it provides access to function information from within the application.

CHAPTER 3
Background: Memory Analysis Tools

In this chapter we investigate some of the principles behind software tools, to understand their methodologies and roles in the software ecosystem. Whilst our focus is primarily on memory analytics many of the concepts presented here are generic and can be applied to different aspects of application analysis. Similarly these techniques are as applicable to parallel software environments as they are to serial environments.

We conclude this chapter with an analysis of a number of different memory analysis tools, evaluating each tool's capability and suitability for memory consumption analysis. From this analysis we identify limitations in the current tool chain, and use this to motivate the development of a new suite of memory analysis tools.

3.1 Types of Tool

In this section we discuss the two fundamental classifications of tools, and their association to applications. By understanding the level of application information required we can make an informed choice on the best data collection methodology to use; we discuss these methods in Section 3.2.

3.1.1 Debugging

Debugging tools are specifically focused on identifying problems with applications, such as the cause of segmentation faults or race conditions. Within the field of memory debugging there is specific focus on identifying array indexing errors and identification of uninitialised values.

The data collection operations required to provide this level of detailed analysis are inherently expensive, so many tools will strike a balance between overheads and complexity of analysis. Such a balance can be achieved by using a mixture of data collection techniques, as this will provide some level of expensive, deep analysis, interlaced with some cheaper, high level analysis.

The process of collecting all the information, for debugging purposes, can facilitate the recording of some incidental metrics. In the case of memory analysis tracking allocations for memory leak detection can also provide some memory consumption statistics at no additional cost. Using tools for these additional metrics is usually an expensive way of extracting the information, but as the features are presented in an existing framework there can be a certain appeal to their usage.

3.1.2 Profiling

Tools for profiling are more concerned with extracting a comparative metric of performance or resource usage. Often profiling tools are of a lighter weight, in terms of overheads, than debugging tools as they generally require less code interruption.

Data collection is usually achieved through strategic instrumentation points; this can be timers placed at regular intervals or queries for resource utilisation. For certain specific profiling operations it is often required to capture the information contained in function calls, through a method of interception.

One concern of profiling methods is avoiding the impact of the act of profiling, both in terms of minimising impact and preventing instrumentation skewing the metrics. In the case of memory consumption this entails ensuring that the memory overhead of the tool used to record memory consumption is not recorded. With performance analysis tools, it is minimising timer overhead and preventing the timing of the instrumentation code.

3.2 Interface Methodology

Both debugging and profiling tools need to collect information about the underlying application to perform their analysis, but the method chosen for data collection is crucial. Different methods will facilitate access to varying depths of information, which may or may not be applicable to the specific tool, but each is accompanied by an associated cost.

In this section we detail a number of the different methods available to tools for data collection, and identify the benefits and drawbacks of each associated method. We note that many production tools will call upon aspects of multiple different data collection methods, to increase portability and improve performance.

3.2.1 API

The use of an API in tools is not particularly common, as it requires source code modifications and will result in a binary targeted towards the specific tool. Such an approach may be use to integrate a low overhead library which will be utilised to capture statistics on every execution, such as the Ichnaea timing library [4].

Some libraries provide profiling APIs, such as the MPI standard profiling interface (PMPI) [62], to allow easy integration of profiling tools. A similar approach is taken by the hardware counter analysis tool PAPI, which then facilitates the construction of other tools which can utilise these function interfaces [78].

Another serious limitation of API based tools is the reliance on calls from the application, relinquishing a certain amount of control to the application. This can dramatically limit the available data if the API is not fully utilised. From a performance aspect the overheads are dependent on the frequency of API calls, enabling a certain amount of performance control within the tool.

3.2.2 Interposition

An extension of the concept of API instrumentation is the use of function interposition, or interception. This is the process of intercepting a call to a library function with one belonging to the tool, facilitating the collection of data before returning control back to the application [26].

This method is feasible due to the way dynamic libraries are loaded with an inherent ‘order’, allowing function calls of the same name to be chained. This allows a tool to intercept a function of a given name, perform a data recording operation, and then call the originally desired function, before returning the result to the application.

One of the main limitations of this method is that tool control is only available during the calling of specifically intercepted functions, and in the case where multiple similar functions exist a matching wrapper function must be written for each. As this method is based on the linking order of dynamic libraries, it does not require the application to be recompiled or relinked; rather, it relies on the runtime linking phase.

Again the performance impact of this method is defined by the number of functions intercepted, and the frequency of their use within the application.

3.2.3 Code Injection / Pre-processing

One of the limitations of function interposition is that only function calls to dynamically linked libraries can be intercepted. Code injection is the process of recompiling the application code but inserting function calls at strategic points, in accordance with the needs of the tools. Such calls could be surrounding calls to important functions, or just at regular intervals.

One good example is the use of OpenMP pragma instrumentation, via pre-processing, in the OPARI source-to-source translator [87].

The process of automated instrumentation allows the tool to gather sufficient information, around certain interpreted sections, simplifying the user experience. As with API based instrumentation the user is left with a binary targeted specifically towards a specific analysis tool, and not suitable for production use as each execution would incur the overhead of the tool. A second binary is thus required, without tool instrumentation, for performant executions. Similarly access to the source code is essential, which is often not the case with ISV applications or libraries.

3.2.4 DBI

Dynamic Binary Instrumentation (DBI) describes the process of inserting instrumentation calls into the underlying application during runtime. The process utilises just-in-time (JIT) compilation to generate, and insert, efficient instrumentation code into the application.

One such system for Dynamic Binary Analysis (DBA) is Intel's Pin framework, which enables the development of tools for DBI [80].

Profiling with DBI based tools does not require application recompilation as instrumentation is done at runtime. The overheads incurred by DBI tools are entirely dependent on the level of instrumentation performed, but it is potentially a very expensive methodology.

Shadow Memory

Within the field of memory analysis, one of the most important operations available to DBI tools is memory shadowing. Shadow memory is a duplicate of physical application memory, but providing an instrumentation interface to memory operations.

This facilitates the storage of meta-data about the contents of the memory, allowing the tool to record information such as access counts, array boundaries or data type. This facility is specifically useful for tools to analyse memory corruption and indexing problems, and the Valgrind suite is a particularly popular framework for the development of such tools based on this technique [93].

The process of shadowing memory is inherently expensive, as it adds additional workload to every memory read and write operation and introduces a large number of context switches. Additionally the storage requirements are also high, as it must essentially duplicate the whole memory space of the application. At each level of analysis, either at the bit or object level, associated meta-data is required and thus the additional memory consumption is determined by the size of this meta-data. These additional memory requirements limit the use of such tools in memory-constrained environments.

The combined performance and memory overheads make shadow memory tools extremely heavyweight, and should only be used to locate specific memory corruption problems, rather than general statistics.

The Umbra [131] scalable shadow memory implementation based on the DynamoRIO platform [17], presents a number of performance optimisations over Valgrind, but the toolset remains limited. They demonstrate a reduction in overheads on the SPEC suite of benchmark applications from 9.47× for Valgrind to 3.11×.

3.2.5 Sampling

Sampling is a slightly different method of data collection to those presented previously, as is it not based on an application operation or function call. Rather, it is based on a temporal interrupt. The concept is to interrupt application execution at regular intervals and gather certain information about the current state of the system.

Such an approach to limited to the data it can collect, as the classification of the function interrupted is not controllable. Whilst it is still feasible to perform a stack trace, the information may not be representative since the chances of striking during an operation of particular interest is often low. Similarly, as the interrupt is handled away from execution flow it is very hard to extract function parameters.

The use of such a data collection technique is therefore limited to establishing statistical probabilities, which may be useful in hot spot analysis or querying a maintained source of information such as hardware counters of system files.

The overheads of this method can be extremely minimal, as the number of interrupts can be very low, but the level of certainty will be directly affected. As such this method is used by lightweight tools to gather a basic understanding of application behaviour, usually as a prelude to more in-depth analysis.

3.2.6 Lightweight and Heavyweight Tools

As we can see from the discussion on data collection techniques there is an obvious divide between lightweight and heavyweight methodologies. Tools which implement these methodologies will inherit their specific overheads, thus to a certain extent defining the cost of the tool.

All of these methods describe how to introduce a point of tool control during application execution, and an inherent measure of frequency. One considerable factor of overheads then becomes the operation at point of tool control. A lightweight tool might simply record a single metric or perform a check, whereas a more in-depth tool may perform a stack traversal or complex validation.

3.3 Related Tools

Whilst memory is a widely investigated topic, the majority of the research todate has focused on performance and correctness rather than capacity. The majority of profiling tools have been developed with memory as their specific point of interest have largely aimed to measure and improve performance, and capacity analysis is a largely unexplored area. Simple analytics are easy to obtain, but provide minimal insight into trends or cause. More complex analytics require a much deeper investigation, one that many non-targeted tools will not attempt.

We classify applications into two distinct camps, lightweight and heavyweight, based on the level of intrusion on the underlying application.

Delistavrou and Margaritis present a comprehensive comparison of many tools for the HPC environment [30]. Their evaluation is based on general capabilities, rather than the memory centric analysis we present here.

3.3.1 memP

memP was developed to understand parallel application memory consumption, and as such comes closest to providing a level of analysis acceptable to drive application improvement. It is designed to capture allocation information as a program is running, and relate this information to provide an overview of memory consumption.

Developed at LLNL as a standalone tool to answer specific questions about the memory consumption of MPI based parallel application, the specific focus of the tool is heap memory, specifically tracking live allocations through function interception. By processing the allocation information the tool is able to calculate the HWM of memory allocations for each processor; this information can then be combined for a more complete overview of the application's memory usage. The design of the tool is to collect and store all information internally, a method inherently flawed for large complex runs where the volume of this information could be detrimental to the performance of the underlying application. The internal storage method also has performance implications, as each allocation-deallocation pair must be matched in order to track live allocations and total memory consumption.

One of the biggest flaws to this tool todate is the inability to store and process this information at a later date, as it is discarded at the end of execution. The loss of such data prevents any deeper analysis than consumption. By only providing HWM data the tool is able to resolve neither where or when the peak consumption occurred, nor the duration.

3.3.2 MAP

MAP is a commercial product by Allinea, the makers of the DDT parallel debugger. MAP is designed as a performance analytical tool which uses sampling, rather than events, to supply information. Whilst this method is less accurate, as key events could be missed, it does provide a very lightweight method of tracking code progression. Although designed to identify computational hotspots, the tool also provides the facility to monitor memory consumption over execution. The information provided is gathered by querying the process information file and logging the number of pages of memory allocated to that process. MAP records the RSS value which, as discussed in Section 2.2.3, is an over prediction due to the handling of shared libraries. This methodology is problematic, due to gaps in the data logging, but is a fast and easy way of visualising a rough description of application memory consumption over time.

The use of sampling in MAP means that whilst the HWM data may be highly accurate there is no context to the data. As the allocations are not tracked the consumption profile cannot be related back to allocations in the code, and this prevents further analysis.

3.3.3 MEMWATCH

MEMWATCH is a memory error checker designed to monitor applications for memory leaks and other memory faults. MEMWATCH does have some functionality targeted at consumption monitoring, but at the time of writing this is underdeveloped and only provides minimal consumption analysis.

Unlike the other tools discussed in this section MEMWATCH requires application recompilation, as code must be linked against one of their header files. This level of recompilation does not suit complex applications with multiple dependencies, such as MPI applications, which must also be also be recompiled. As such, we do not evaluate MEMWATCH further as a viable tool for large scale parallel application analysis.

3.3.4 Valgrind

The Valgrind suite is a slightly more complex case, as it is a framework for the development of tools, rather than a specific tool [93, 94]. Despite this, there are some well established tools which provide some focus on memory consumption.

A significant portion of the focus for the Valgrind tools is the debugging of applications. These tools are traditionally aimed at serial applications, but parallel applications are are supported. The type of bugs the tools are designed to identify are memory problems, often where memory outside of the normal frame of use is referenced. To achieve this Valgrind employs the use of shadow memory at high computational cost. This makes the overheads of Valgrind based tools non-trivial, and their use to extract memory consumption data excessively expensive. We also note that due to the way Valgrind occupies memory space there can be issues profiling binaries with large statically allocated memory segments, if these address regions overlap the instrumentation will fail.

Memcheck

Memcheck is the default Valgrind tool, which is designed to monitor for improper memory usage, such as array index out of bounds errors and un-initialised memory. In addition to error checking, Memcheck returns some select statistics on heap consumption, such as the HWM.

Massif

Massif is targeted much more specifically at memory consumption on the heap, and provides more statistics regarding consumption. Massif works on the idea of 'snapshots' that record the current state of memory, which is possible due to their internal memory management. From these snapshots Massif can then generate consumption graphs, and even functional break downs. This functionality is very desirable, but unfortunately is presented in a very heavyweight framework thus limiting its appeal.

3.3.5 Existing Tool Critique

Whilst there are a number of memory analysis tool already providing certain capabilities they are not designed to provide detailed non-intrusive low-level analysis.

Those applications which do provide low-level analysis do so at a prohibitively expensive cost, due to their data collection methodology. Whilst those that employ less intrusive data collection methodologies fail to expose the level of detail required by the user to really understand what is happening within the application.

3.3.6 Motivating New Tools

It is clear that there is a gap in the market of memory analysis tools, there is a need to collect detailed memory consumption statistics and provide detailed analysis, within a lightweight framework.

Massif is the only other tool to begin to fully analyse the cause of memory consumption problems, and it does so in a prohibitively expensive framework. Additionally the analytics are relatively basic, given the depth of information available to the tool.

We aim to develop a new tool which expands upon some of the ideas discussed in this section, to provide greater data collection and analysis than existing tools. Additionally we aim to do this in a lightweight and non-intrusive environment, facilitating the memory analysis of production applications.

3.4 Summary

In this chapter we have introduced some of the key principles and methodologies of software tools. By describing the different classifications and their data requirements, we have been able to compare and contrast different data collection techniques. Understanding the inherent benefits, and associated costs, of these techniques allows us to understand the benefits and drawbacks of tools based on these techniques.

This emphasises that tools based on shadow memory techniques will be able to provide a very deep level of data collection and facilitate through analysis. However this methodology will incur significant performance detriment which, whilst acceptable for debugging tools, is excessive for tracing tools. Function interposition tools can capture similar volumes of memory allocation data when compared with shadow memory tools but are unable trace memory usage. This level of data collection is available for a significantly reduced computational overhead, making this class of tool more viable for tracing. Sampling based tools provide a much more restricted level of data collection and associated analysis than the other classes of tool but have a much lighter profiling overhead. This makes such tools a good first step in profiling, allowing the user to identify problems, but are often unable to back up this identification with analysis.

Our evaluation of these existing tools is then used to identify a gap in the memory analysis ecosystem, thus motivating the development of a new memory consumption analysis tool. By collecting data via the function interposition method, like memP, but storing this allocation data with stack trace information, like Massif, we can perform a varied array of deep analyses offline, thus minimising the runtime overheads of the tracing tool.

CHAPTER 4
WMTools

In this chapter we discuss the development of our own memory consumption analysis tool suite: WMTools.

Based on our analysis of existing tools in Section 3.3 we develop a multi-component tool suite designed for increased analytics, where applications first require a data collection phase and then a data analysis phase. We document the specifics of the different components of the suite, and the way they combine to provide analytics. We demonstrate the different forms of analytics through a case study of the AWE Chimaera benchmark.

To conclude, we provide an analytical comparison of WMTools with the previously discussed memory tools, identifying functionality differences, and comparing performance.

The majority of the research presented in this chapter was originally published in [106], and the concepts developed further in [103].

4.1 WMTrace - Data Collection

WMTools is designed on the principle of data collection and retention. WM-Trace is the tool specifically designed to collect memory allocation information from parallel applications and save this information to file, allowing for in-depth processing at a later date. Whilst the implications of this data capture and storage methodology are non-trivial the benefits are clear. Despite potentially gigabytes of trace information, representing millions of nodes in the application call tree, being generated you have an exact replica of the series of events allowing for in-depth analysis [39].

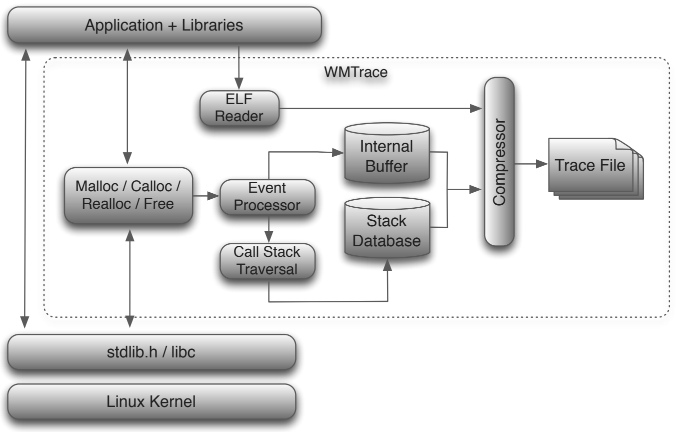


Figure 4.1: WMTrace data collection process

4.1.1 Library Structure

WMTrace is a dynamic C++ library which interposes POSIX based memory allocation calls, such as malloc, calloc, realloc and free.

Figure 4.1 illustrates the internal layout of WMTrace. As we can see the library sits between the application and other dynamic libraries such as system libraries. Data from memory management calls are intercepted and passed to the event processor which records the size, time and location of allocations. This event data is then written to an internal buffer. Call stacks, which are generated from these events to represent the location of an allocation, are passed to a stack dictionary which maps call stacks to a unique ID, as a mode of compression. Periodically the internal buffer is flushed and along with a list of the newly observed call stacks this information is passed through a compression engine, which in-turn passes the data to file. Analysis of the application, through Elf and the virtual address space, is also performed and stored in the trace files, allowing function addresses to be resolved at a later date.

4.1.2 Application Interaction

WMTrace is a dynamic library which is linked via an LD_PRELOAD operation at runtime, during the application setup phase. There are many benefits to this format, including avoiding the need for compile time linking – there is no need to recompile applications before tracing them with WMTrace.

WMTrace is specifically designed to handle MPI based applications, and is initiated by an application's call to 'MPI_Init'. This allows WMTrace to establish separate trace files based on rank information.

ELF

From the binary WMTrace is able to ascertain the static memory partition, which doesn't present as a malloc but still contributes to memory consumption. WMTrace also queries the ELF header for function address information, this is used to resolve addresses obtained during call stack traversal. To gather information regarding the function addresses of dynamic libraries, WMTrace must query the virtual address space, using the 'dl_iterate_phdr' function.

We note that WMTrace uses function address information from the ELF headers and resolves locations to within function address ranges, as such information is largely available even without debugging information in the binary.

Stack Tracing

Stack tracing allows WMTools to understand the 'location' of an allocation, with respect to the sequence of function calls which caused it. This information is essential for any form of complex analysis that differentiates between allocations. However the collection of this information is expensive, and can generate a lot of data. The complexities are handled by a third party stack tracing library, libunwind [90], which is reasonably efficient and highly portable.

We experimented with alternative methods of collecting call stack information. There are various methods of improving the performance of frequent call stack traversal, using additional operations [40, 92, 110, 119, 121, 125]. Many of these methods involve modification of the stack, and the insertion of markers, allowing for detection to prevent further unnecessary traversal.

We developed a heuristic call stack traversal method, presented in [104], which uses the repetition of patterns and the stack size to deduce change. With this method we were able to predict call stack suffixes with an average accuracy of 89%, providing an overall speedup of 12% to WMTrace.

Using our initial technique some applications, such as AMG, experienced stack prediction accuracy as low as 5.2% – a result of low call stack densities within the application. Methods employed to improve this accuracy were detrimental to the performance of the technique, thus reducing the gains available to WMTools.

During this heuristic traversal we were unable to validate our predictions without knowledge of the correct call stack information. Thus the variability of accuracy becomes an issue, as this would inturn diminish confidence in later analysis we did not pursue the method any further, within WMTools.

4.1.3 Data Storage

WMTrace has a simple method of data storage, utilising a single trace file per MPI process. This allows each process to act independently, saving runtime, but resulting in potentially large combined file output.

Data storage is key to WMTools, as it facilitates the offline analysis of runs, allowing for different forms of analysis to be performed as and when they are required. The drawback of this method is the volume of data generated, with implications on both storage and I/O performance.

As WMTrace therefore employs lossless data collection and storage, the size of the trace files is dependent on the number of allocations, which in most circumstances will grow over time. The implication is that with extremely long runs these trace files will build up in size, potentially causing problems.

WMTrace employs an internal buffer, to facilitate the periodic staging of data to file. As this buffer is appended to file it is passed through a ZLib [133] compression engine, reducing the data volume.

Ferreira *et al*. discuss the importance of managing data storage volumes in HPC, and the potential benefit of using standard compression algorithms to minimise data from log outputs [38]. They utilise a parallel pzip2 algorithm, pbzip2 [44], in conjunction with a staging area, similar to the internal buffers utilised in WMTrace. They achieve compression ratios of over 80% HPCCG (a conjugate gradient benchmark), though as the compression was handled by a dedicated 'spare' core they do not discuss the performance implications of this technique.

WMTrace handles the storage of stack traces in a different way to events. As there is a lot of repetition, we maintain a map structure recording all unique call stacks. This method of compression is more efficient than relying on ZLib to spot repetition. Newly observed call stacks are then periodically written to file, before the event trace segment, and are passed through the ZLib engine for additional compression. As a form of fault tolerance trace files are well formed, allowing the partial analysis of runs which fail mid execution.

4.2 WMAnalysis - Data Analysis

With memory consumption analysis, the most fundermental metric is high water mark (HWM). This value will determine if a job will fit in the available memory of a compute node, thus is a crucial metric.

From analysis of HWM values, we can gain a high-level understanding of memory scalability. To interpret the memory consumption behaviour of the application we need to undertake much deeper analysis. In this chapter we present a number of analytical methods, available in WMTools, and discuss the ways in which they can help code custodians.

To demonstrate the capabilities of the different analysis methods we illustrate this section with a case study performed on the AWE application Chimaera. We will use each analysis method to investigate a different facet of the application memory profile, to gain insight into the application. We base these analytics on application runs performed on the LLNL Cab platform, based on a strong scaling study to represent normal usage.

4.2.1 Analysis Phase

The analysis phase, provided by the WMAnalysis tool, can be executed in different ways depending on user requirements. The analysis can be performed as a distinct operation on the trace files, which can happen at any point after execution – even on a different platform. As the processing on multiple files can be expensive, WMAnalysis can make use of a parallel environment, such as that belonging to the initial job, to perform some analysis. As such a post-processing phase can be triggered to occur at the end of application execution, on the newly acquired data.

Our preferred method of analysis is to use this parallel post-processing phase to undertake a very high level HWM analysis, to establish the ranks of particular interest. Then we undertake an offline analysis of these specific trace files with the more comprehensive analysis tools. This method enables us to delete files of little interest whilst preserving those of interest for further analysis, freeing up disk space.

4.2.2 HWM Analysis

HWM analysis is the the most fundamental analytical form in WMAnalysis, simply reporting the HWM value and the static memory consumption. When run in parallel, WMAnalysis will report statistics on the full job, such as maximum and minimum HWM, and their respective ranks, in addition to a measure of standard deviation. This helps gauge if there is a memory imbalance between the processes, which may have resulted from a poor workload decomposition.

```
# Max mem - 288269798(B) (Rank 48)
# Min mem - 245339398(B) (Rank 127)
# Standard deviation - 1.4684e+07(B)
# Static memory consumption of 10985316(B).
```

Listing 4.1: Chimaera HWM report for 120^3 on 128 cores

	Runtime (s)	Max Mem (MB)	Max Rank	Min Mem (MB)	Min Rank	Standard Deviation (MB)
16	56.82	230.41	0	227.94	15	0.57
32	37.66	137.45	0	118.68	31	7.09
64	28.29	88.70	0	61.98	60	6.54
128	28.39	65.72	0	33.66	103	6.40
256	43.06	61.76	0	18.87	255	7.35

(a) 60^3 HWM profile

	Runtime (s)	Max Mem (MB)	Max Rank	Min Mem (MB)	Min Rank	Standard Deviation (MB)
32	228.33	860.30	0	853.72	31	1.32
64	131.68	457.06	16	449.56	63	1.69
128	87.21	274.92	48	233.97	127	14.00
256	85.16	179.54	0	122.07	207	12.58
512	199.86	148.23	0	66.32	399	13.21

(b) 120^3 HWM profile

Table 4.1: Chimaera HWM strong scaling profiles

Chimaera HWM

When WMTools is run with the instruction to post-process it generates a small HWM report at the end of execution, utilising the parallel environment of the job to execute WMAnalysis in parallel. Listing 4.1 illustrates an example of the WMAnalysis output at the end of execution.

From Table 4.1 we can see both the runtime and memory profile of Chimaera running the 60^3 and 120^3 problems. We note that the 512 core 60^3 and 16 core 120^3 runs were unable to execute due to invalid processor decompositions and insufficient node memory, respectively. From these tables we can clearly see that the HWM scaling of Chimaera is fairly poor at this problem size, but we also see that there is very good scaling for the minimum HWM process. This is indicative of a workload imbalance, and some rank specific behaviour such as file I/O or communication.

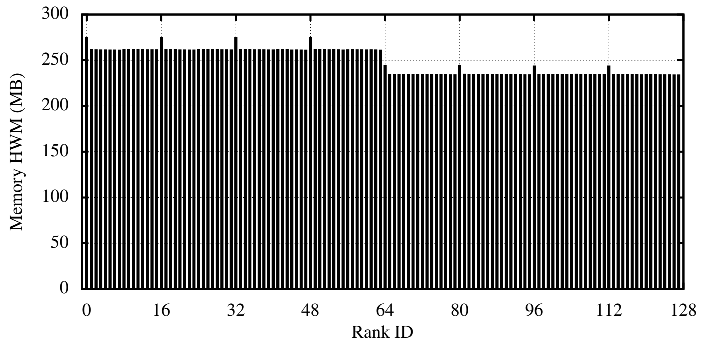


Figure 4.2: Per rank HWM distribution for Chimera 120^3 on 128 cores

Additionally we note that the application exhibits poor runtime performance at large scale. This is a result of the inefficiencies of wavefront codes when operating on small ‘pencils’ of data. In the case of 120^3 on 512 cores the 2D decomposition results in a 7×3×120 ‘cube’ which is naturally inefficient for a wavefront code to calculate.

To better understand this HWM imbalance, we plot the HWM values of each thread in the job. Figure 4.2 illustrates this for the 128 core run of the 120^3 problem. We can clearly see two artefacts in this figure. The first is the average disparity between the first half (64 cores) of the job and the last half. This disparity of ≈27 MB is most likely to be the impact of a workload imbalance. Chimaera utilises a 2D processor decomposition, resulting in a 16×8 decomposition for 128 cores. When applied to the first two dimensions of a 120×120×120 cube we are left with an imbalance; 64 processors have a 15×8×120 sub-domain and the remaining 64 processors have a 15×7×120 sub-domain. If we allow for a ghost cell halo, we obtain sub-domain sizes of 17×10×122 and 17×9×122 respectively, which gives a ratio of $\frac{10}{9}$ between the first and last 64 ranks.

The second artefact is the repeating increase of ≈13 MB every 16 cores. This repetition correlates with the 16 cores per node structure of the Cab platform, suggesting that a single nominated rank per node is adopting a certain operation, most probably file I/O or additional communication.

4.2.3 Functional Breakdown

Analysis of HWM values can reveal artefacts in consumption but provides little explanation of the cause. In addition to HWM values WMTools is able to generate functional breakdowns, allowing the user to perform in-depth analysis of the allocations live at the point of HWM.

By recording the call stack of each memory allocation, WMTools is able to group allocations, by location, so as to see the memory consumption from each area of code. This enables the interrogation of HWM allocations for specific functions or libraries. In Chapter 6 we exploit this feature to monitor the memory consumption of the MPI library at the point of HWM. This allows us to closely monitor changes in MPI memory consumption that occur in different runtime configurations.

Chimaera Functional Analysis

```
# HWM Functions file from WMTools - WMTrace/trace-48.functions
#       HWM of 288269798(B)
#
# MPI Memory summary: 14124864(B)
#       (4.89988%) of memory attributed to MPI (libmpi)
#
# High Water Mark Function Breakdown

Call Stack: 5513 Allocated 212377600(B)
        (73.6732(%) ) from 128 allocations

for_alloc_allocatable
-setup_storage_flux_mod_mp_setup_storage_flux_
--chimaeramain_
....
```

Listing 4.2: Chimaera HWM breakdown report for 120^3 on rank 48 of 128

Listings 4.2 and 4.3 present the start of the HWM function breakdown analysis for our high (rank 48) and low (rank 127) HWM processes for the Chimaera 120^3 problem on 128 cores. We present the header and the first entry in the function list, representing the highest consuming call stack.

By comparing the MPI consumption we can clearly see that the maximum HWM process, rank 48, contains ≈13.5 MB of MPI memory, whereas the minimum HWM process, rank 127, only contains ≈0.1 MB. This ratifies our

```
# HWM Functions file from WMTools - WMTrace/trace -127.functions
#        HWM of 245339398(B)
#
# MPI Memory summary: 148578(B)
#        (0.0605602%) of memory attributed to MPI (libmpi)
#
# High Water Mark Function Breakdown

Call Stack: 5299 Allocated 191139840(B)
        (77.9083(%) ) from 128 allocations

for_alloc_allocatable
-setup_storage_flux_mod_mp_setup_storage_flux_
--chimaeramain_
....
```

Listing 4.3: Chimaera HWM breakdown report for 120^3 on rank 127 of 128

explanation of the 13 MB difference occurring periodically on the first rank of each node illustrated in Figure 4.2.

Another comparison is the size of allocations made by ‘for_alloc_allocatable’, as both threads exhibit the same number of allocations. The ratio of these two sizes, 212377600 B and 191139840 B, perfectly matches our proposed problem size ratio of $\frac{10}{9}$, when assuming ghost cells. Additionally both of these sizes perfectly represent an allocation of 10240 B per local cell.

We note that whilst this mathematical observation accounts for the largest memory consumer at this scale, accounting for over 70% of HWM consumption, WMTools allows us to track all other points of consumption. The tool also enables us to observe how these percentages of consumption change over time, due to the memory scalability of different components.

4.2.4 Temporal Graph

The HWM metric is a single value for a process, revealing the point of highest memory consumption. What this figure does not reveal is for what percentage of time this HWM level was sustained. The natural progression is to understand the variance of consumption, if the HWM value is significantly higher than during the remainder of the application’s execution.

A high variance in HWM suggests that there is a single point of execution where a large volume of memory is allocated, shortly followed by memory deal-

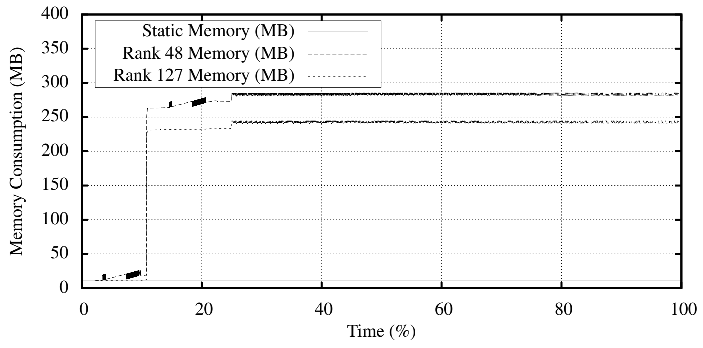


Figure 4.3: Comparison of maximum and minimum HWM threads for Chimaera 120^3 on 128 cores

location. Such an event might occur when large buffers are used to temporarily store values during a data manipulation procedure. When the memory HWM occurs as a result of a memory inefficient procedure, the implementation of an alternative method for a potentially trivial operation, could have dramatic influences on memory HWM.

Such is the case with matrix transpose, where data can be duplicated to represent it in a different order. The use of in-situ matrix transposition allows small blocks to be transposed with minimal buffers and without the allocation of a second matrix [45].

By visualising memory consumption with a temporal reference point, it is clear to see whether memory is fully utilised, and where there is potential for optimisation. The comparison of different temporal traces then gives further comprehension of the relationship between threads within the same job, and between jobs of different size.

Chimaera Temporal Graph

Utilising the same 120^3 128 cores problem we explore the temporal memory consumption of Chimaera. Figure 4.3 illustrates the memory consumption of the maximum and minimum HWM ranks, 48 and 127 respectively. From this figure we can clearly see that there is a very regular pattern to the memory consumption, replicated on both ranks, where the consumption starts low until about 12% through execution where it spikes, it further settles at 25%, for the duration of the run.

By comparing the two traces we can see the absence of the minor consumption growth at around 10% on rank 48, and the further variation at around 20%. Again, this behaviour is indicative of rank specific operations.

4.2.5 Temporal Function Graphs

To expand upon the idea of temporal analysis we combine it with the functional breakdown discussed previously. This enables the visualisation of the composition of the memory consumption during execution.

By looking at functional consumption at HWM it is hard to grasp the scope of the allocations. How long has a contributing function had that memory allocated? How long will that consumption remain? By answering these questions we can provide more insight into potential memory consumption optimisations.

As the number of unique call stacks is far too large to visualise, we restrict our analysis to single unique functions. For this we use the last function in the call stack, before the allocation. Using the breakdown of function consumption at the point of HWM we select the top consuming functions, ordered by contribution to HWM. All remaining functions are grouped into an additional 'other' category.

This form of analysis relies on sampling of the WMTools trace files, so as to minimise the resulting output graph size. The number of samples, and functions to display, is user configurable at analysis runtime.

These functional graphs allow the quick comparison of different trace files, to monitor the cause of differences.

Chimaera Functional Graph

Figure 4.4 plots the memory consumption of the two most dominant contributors, the function 'for_alloc_allocatable' and the MPI library 'libmpi.so', for

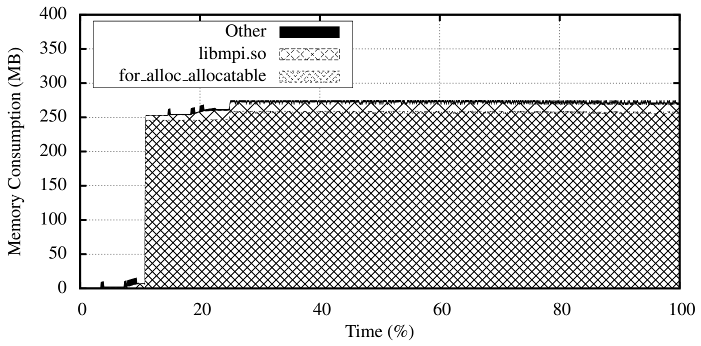


Figure 4.4: Rank 48 temporal functional graph for Chimera 120^3 on 128 cores

the maximum HWM thread, rank 48. Listing 4.2 illustrated the dominance of these functions at the point of HWM, but Figure 4.4 illustrates how their consumption changes over time.

We can see that the MPI memory consumption remains fairly constant, after an initial ramp up in the first 25% of execution. Additionally we see that the other contributing functions, collectively referred to as 'Other', only consume a very small portion of total memory, but are responsible for the initial variance at around 12% through execution.

4.2.6 Heat Map

Many of the previous analytic methods present a single metric, such as HWM, or the progression of a metric over time. These methods focus on a single process. With the heat map analysis we wanted a way to compare the values of the processes over time.

WMHeatMap takes each trace file and samples the output for memory consumption over time. For every time sample we can then represent the current memory consumption of every process in the job. Using the Silo data format [69], developed at LLNL, we generate VisIt visualisations [70], which allow this data to be represented on a grid, over time. Where available, we use the processor decomposition of the original job to represent the memory data, with processes

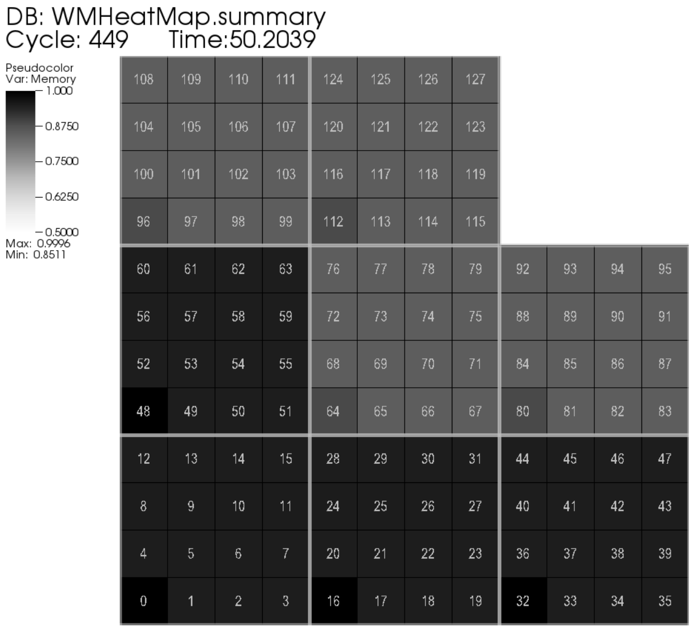


Figure 4.5: Heat Map at point of HWM for Chimera 120^3 on 128 cores

grouped to nodes.

The benefit of this analysis method is it allows the simultaneous visualisation of the temporal graph for every process. Additionally, we are able to identify areas of localised memory consumption and consumption imbalance. Much like node-level consumption discussed previously, we are interested in the total memory consumption of a node; WMHeatMap provides an intuitive way of visualising this data.

By grouping the processes by node it is easy to identify patterns in artefacts. Poor workload decompositions may result in clusters of high memory consumption processes. Equally, patterns in MPI memory consumption become very easy to spot.

Chimaera Heat Map

Figure 4.5 shows a snapshot of WMHeatMap displaying the rank level memory consumption of Chimaera at the sampled interval point closest to the HWM.

For visual clarity we have utilised a greyscale mapping from black at 100% of HWM, 274.92 MB, to white at 50% of HWM. As we can see from the figure there is a dominant memory consumption on the first four nodes of the job and a lower consumption on the remaining four. We can still make out the increased consumption in the first rank of each node, as the rank grouping has a clear representation rather than our previous assumptions.

The information portrayed in this figure is very similar to that of Figure 4.2, for this particular example. If we study the temporal graph in Figure 4.3 we see that Chimaera has a very flat memory consumption whilst at peak. This means that whilst the HWM of particular ranks may not occur at the same time, the magnitude of their consumption will be close to their HWM at any given point during the ≈22% to 100% phase of execution. With an application which exhibits a more sporadic memory profile, the variance will result in a different representation.

4.2.7 Comparative Analytics

In memory analysis we are often concerned with absolute value of consumption. For example, if one thread has a higher HWM value than another, we can draw conclusions. What is less obvious is how comparable these values are if they occurred at different points in execution. Comparison of functional consumption at HWM may not hold as much significance as initially believed, if the HWM occurred at different points in time for the different threads.

For this reason we wanted to develop a methodology for comparing threads at equivalent points of execution.

This process involves being able to analyse a process at any arbitrary point within execution, not just the point of process HWM. Thus we are able to sample the processes of a job at multiple time points, and analyse consumption at key events.

When comparing between jobs, this act becomes significantly more difficult, and we must look to application behaviour to establish the correct time.

	HWM (MB)	Time (%)
1	4196	92.2
2	4196	92.2
3	4195	92.2
4	4196	92.2
5	3759	92.2
6	3760	92.2
7	3761	92.2
8	3756	86.4

(a) 128 cores

	HWM (MB)	Time (%)
1	2331	83.5
2	2334	74.4
3	2332	83.3
4	2333	83.3
5	2326	70.8
6	2325	83.5
7	2325	83.3
8	2326	83.5
9	2085	55.2
10	2083	83.3
11	2082	89.1
12	2083	83.5
13	2081	80.0
14	2082	56.4
15	2083	47.6
16	2083	57.0

(b) 256 cores

Table 4.2: Chimaera node level HWM for 120^3

Chimaera Node Level HWM

The version of Chimaera used in this study does not make use of OpenMP parallelism, only MPI. We therefore use our comparative analysis to investigate node level memory consumption for the code. Due to thread HWMs occurring at different times the node level HWM is not necessarily the sum of each process HWM. Rather, this only forms a theoretical maximum.

In Table 4.2 we present the node level HWM values for Chimaera 120^3 on (a) 128 and (b) 256 cores. From Table 4.2(a) we can still see the distinct difference between the consumption of the first and last 64 cores, due to imbalance of problem decomposition. If we study the node-level HWM between these two runs we can gauge the memory scaling, as opposed to the rank-level scaling observed earlier. We observe a 1.8$\times$ reduction, over the ideal reduction of 2$\times$ for a halving of local problem size as we transition from 128 cores to 256. If we compare this to the rank level HWM reduction, from Table 4.1(b), we see a 1.5$\times$ reduction. Thus we can conclude that the node level memory consumption has better scaling than the rank level. As memory is traditionally provisioned at a node level we consider this to be the more crucial relationship.

	WMTools	memP	MAP	memcheck	Massif
HWM Value	✓	✓	✓		✓
HWM Time	✓		◇		✓
Stack Tracing	✓	✓		✓	✓
Temporal Graphs	✓		✓		✓
Memory Leak Detection				✓	✓
Inherently Parallel	✓	✓	✓	◇	◇
Data Maintained	✓		◇	◇	✓

Table 4.3: Memory tool feature comparison

We do, on the other hand, notice the compound effect of the grouping of ranks with similar decompositions. As we saw in Figure 4.2, the first 64 cores of the 128 core run have a higher HWM than the latter, as these cores are physically located on the same nodes, the node-level differences are exacerbated.

4.3 Comparison

In this section we present an evaluation of tool capabilities, comparing our newly developed WMTools suite with the memory tools analysed in Chapter 3. We compare functionality and identify where capabilities were available in existing tools and where we have introduced new capability. Additionally we clearly identify where WMTools does not provide capability, specifically in the debugging space. Where there is similar functionality we provide some comparable examples of analysis from these other tools.

Additionally we provide a comparison of runtime overheads based on the execution of the Chimaera 120^3 problem application on 64 cores of the Cab platform, allowing us to validate our design goal of creating a lightweight tool. Where applicable we also compare the HWM values of these tools as a measure of accuracy.

4.3.1 Functionality

Table 4.3 compares the functionality of WMTools to four existing tools. In the table a '✓' represents full compatibility and a '◇' indicates partial compatibility.

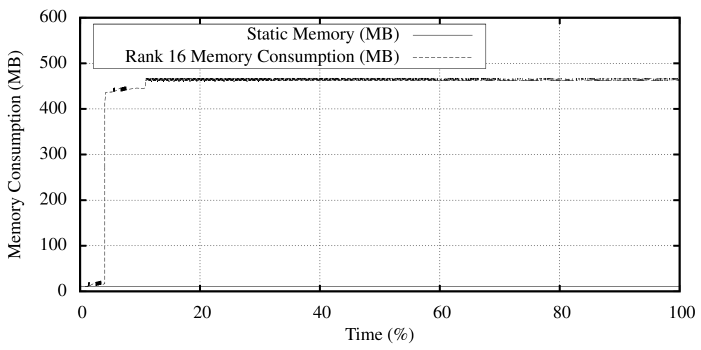


Figure 4.6: WMTrace analysis of Chimaera on 64 cores

We define a tool as inherently parallel if it was originally designed for MPI based parallelism. We also define a tool to maintain data if there is binary output from the tool which can then be analysed post execution, although we exclude text based summaries from this criteria.

Where there is only partial capability for a function we exclude it from this table but discuss it further for the specific tools.

This table illustrates that we succeeded in our original goal to provide a rich feature set, with enhanced analysis, when compared to the other tools available.

WMTools

Figure 4.6 represents the temporal memory trace from WMTools for a 64 core run of Chimaera 120^3, for the HWM rank. We use this for comparison with other tools capable of generating temporal traces.

We note that whilst leak detection is not currently available in WMTools, there is sufficient information contained within the traces to perform some level of analysis, should such an analysis be desired.

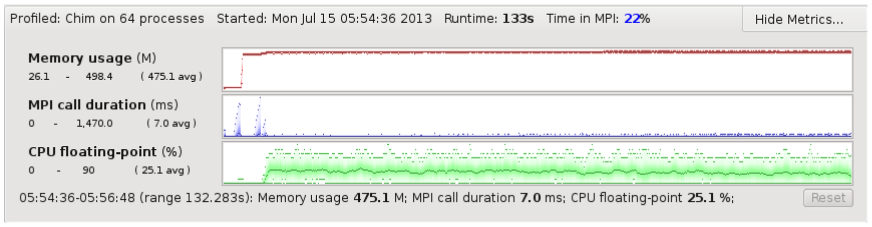
memP

Listing 4.4 represents the output from memP when analysing Chimaera. The similarities between memP and WMTools are reflected in this output, when compared with Listing 4.1, as much of the same information is available.

```
---------------------------------------------------------------------------
@--- Greatest Heap High Water Mark (top 1, descending, bytes) ------------
---------------------------------------------------------------------------
      Rank            Heap HWM     Stack             Sum
         0           479541133     10240         479551373

---------------------------------------------------------------------------
@--- Heap HWM Statistics --------------------------------------------------
---------------------------------------------------------------------------
Max                          :  479541133
Median                       :  472251153
Mean                         :  472642340
Min                          :  471628953
Stddev                       :  1710734
Coefficient of variation     :  0.003620
```

Listing 4.4: Chimaera HWM report from memP for 120^3 on 64 cores



Profiled: Chim on 64 processes   Started: Mon Jul 15 05:54:36 2013   Runtime: **133s**   Time in MPI: 22%

05:54:36-05:56:48 (range 132.283s): Memory usage **475.1** M; MPI call duration **7.0** ms; CPU floating-point **25.1** %;

Figure 4.7: MAP analysis of Chimaera on 64 cores

Although the data collection methods of the two tools are very similar, the feature set provided by them is quite different, as see in Table 4.3, with WMTools providing a much richer analysis framework.

By default memP does not perform stack tracing, and only reports top level HWM analysis. If a deeper level of analysis is required it must be specified to memP at runtime, through environment variables, due to the fact that no data is preserved.

MAP

Figure 4.7 represents a screen shot from the Allinea MAP tool profiling Chimaera. In addition to showing memory consumption we see the application profile in terms of MPI calls and floating point intensity, which helps us to understand the rough composition of the application. Whilst MAP does provide capabilities to 'drill down' into source code, this is done to analyse computation hotspots, and no additional memory consumption analysis is provided.

Although MAP uses a different method of data collection to WMTools we can see that the temporal trace generated is very similar to that of WMTools, shown in Figure 4.6.

We note that MAP is used here in 'profile' mode, where a trace file is generated during execution which can then be loaded into MAP at a later date for analysis.

memcheck

Whilst memcheck records information regarding memory allocations and leaks, it is not specifically designed to monitor heap usage; such functionality is reserved for Massif. It does report the heap memory consumption at point of exit, useful for identifying outstanding allocations, which for the Chimaera application is fairly representative of HWM (as seen in Figure 4.6).

memcheck produces a lot of text output with regard to memory errors, and warnings, but does not store any allocation information for post-processing. Whilst we do not consider this maintained data (as it is of little use for analysing memory consumption) we do record it in our performance comparison in Table 4.4, for clarity.

Whilst Valgrind-based tools are not inherently designed for MPI-based parallel applications, there is sufficient support to enable analysis. One limitation occurring from this lack of inherent support, is that there is no rank information embedded in the output and processes are only referred to by their process ID, which is hard to trace back to MPI rank composition.

Massif

Figure 4.8 represents the ASCII temporal trace printout from Massif, depicting the same execution we represented in Figure 4.6, for the HWM rank. The most significant distinction to make is the graph's x-axis, which is displayed in terms of giga-instructions (Gi). Due to the significant overhead of Massif, it would be unreliable to plot a graph in terms of time, as the act of instrumentation would

```
MB
475.7^                          ::::::: :::::::::@  ::::  ::::::::::::  :::::@
     |#:::::::::::::::::::::::@@:: : : ::  :::: :@::::: ::: : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
     |#: :::: :: : :: : :::: :@ :: : : ::  :::: :@: ::: : : : : ::: ::::: :::@
   0 +----------------------------------------------------------------------->Gi
     0                                                                   646.4
```

Figure 4.8: Valgrind Massif temporal graph for Chimera 120^3 on 64 cores

skew the perspective too much. Although instructions are a good substitute for time, we attribute the differences in the shape of the graph to the irregularity of instructions in MPI applications [108]. The MPI in Chimaera, which is dominant in the early phase of execution (visible in Figure 4.7), will naturally introduce synchronisation and will affect the rate of instructions, hence warping the graph.

In this example Massif collected 88 snapshots, and provides detailed functional breakdowns at 7 of them.

As we can see from Table 4.3 Massif has the closest feature set to WMTools, although the data collection methodology is very different, and this is reflected in the performance comparison in Table 4.4.

The use of process based information in Massif, as with memcheck, makes it hard to understand the composition of the parallel environment.

The primary distinction between the capabilities of Massif and WMTools lay in the way data is maintained, and subsequently presented. The use of 'snapshots' in Massif mean that only a small subset of information is available, at points which it has deemed to be important. Due to the way WMTools maintains an exact copy of the event sequence, all information is available for all points within execution. The benefit of this methodology is it allows the analysis of allocates occurring outside of the defined points of interest presented

	Runtime (s)	Slowdown (×)	Data (MB)	HWM (MB)
Chimaera	131.68	n/a	n/a	n/a
WMTools	165.77	1.26	559.09	457.06
memP	179.04	1.36	n/a	457.32
MAP	132.30	1.00	0.76	498.40
memcheck	3433.91	26.08	32.83	459.96
Massif	1322.27	10.04	1.96	475.66

Table 4.4: Memory tool overhead comparison for Chimaera 120^3 on 64 cores

in Massif. This ability to perform arbitrary analysis provided much more information and facilitates the comparative analysis of multiple trace files, where key points of interest may not have occurred at the same time.

4.3.2 Performance

In Table 4.4 we present an overhead comparison for our selection of alternative memory tools. Additionally we document the size of any trace files, and the recorded HWM value. We note that due to the differences in the way these tools collect their allocation data, there will be differences in the recorded values. The value record by Massif is considered to be the most 'accurate', as it has taken the most information into account, without over-predicting the consumption.

WMTools records the lowest HWM values out of all the tools, though very close to memP, which suggests that there is some consumption which is not being taken into account. We observe that this figure does not take into account the additional ≈10 MB of static memory contained within the binary, reported in Listing 4.1.

The sampling based method, MAP, has the lowest overhead. However, it also over-estimates HWM due to taking its measurements from the RSS value rather than allocation data.

The two function interposition tools, WMTools and memP, have the next lowest overheads, and these are comparable between the two tools. We note that memP has no storage requirements, as data is not maintained, as opposed to WMTools which has comparatively high storage requirements. As such memP may be a preferable choice where data storage is constrained.

Lastly the two shadow memory tools, memcheck and Massif, have the highest overheads with memcheck significantly slower than Massif. We attribute this to the instrumentation performed at every memory operation to check for errors.

These performance trends are in line with those predicted in Section 3.3, and are inherently tied to the data collection method employed, as discussed in Section 3.2.

With regards to data storage as the application in scaled – all tools with data output will see an increase in storage. MAP make an effort to merge similar data across multiple processes, so will see a slower rate of data growth as the core count of the job is increased. Both WMTools and Massif store a distinct file per process, and so the total volume of data output will scale, roughly, proportionally with job core count. Due to the high data volume of WMTools this may present a limitation at large core counts. Both MAP and Massif also take effort to store data in 'snapshots', as such their storage requirements will not vastly vary with execution duration. Due to the lossless approach of WMTools all data is maintained so execution duration will have more impact than for the other lossy tools.

Trace File Size

In prior research we have discussed the storage implications of WMTools trace files, as job core count is scaled [103]. We observed that whilst trace file sizes do increase with core count the relationship is not directly proportional. Across a selection of nine benchmark applications we experienced an average trace file increase on 1.7× for a doubling in core count. Individual applications exhibited average trace file increased of between 1.2× and 2.5× when the core count is doubled.

This mixed behaviour is a result of application structure, as trace file size is determined by number of allocations made and the depth of the call stack at point of allocation. For applications with an allocation count proportional to the local problem size, the trace file size will remain roughly consistent. Applications with a near constant number of allocations the trace files will increase in size proportional to the core count. Increases in allocation count can occur when a job is scaled as a result of the memory management in MPI.

MAP executes a trace merge phase where the individual trace files are merged into a single files, removing the natural duplication which occurs from similar events on multiple processes. This technique could be employed within WMTools to reduce the overall trace file size, but has not not currently been explored.

4.4 Project Availability

This project was always intended to be an open-source tool, available to anyone with an interest in understanding memory consumption. As such WMTools is hosted on a public repository with free access (GitHub) [101], and is made available under the GNU General Public License (GPL).

Whilst no facility exists to track downloads and usage we have spoken to individuals who have tested the software at a number of different institutions.

4.5 Summary

In this chapter we have presented our own memory consumption analysis tool suite, WMTools, after identifying the limitations of existing tools. We have detailed how the different components of the tools operate, making a clear distinction between the data collection and data analysis phases. We have presented a number of analysis forms which exist within the suite, and illustrated an example of these analytical methods.

We have discussed what functionality is already available through existing tools, and illustrated where we have established new functionality. To ensure competitive performance we have also presented an overheads comparison, demonstrating where WMTools ranks in relation to the other tools available. From this we were able to illustrate the enhanced analytics available through WMTools, at comparable and often favourable levels of overhead to existing tools.

CHAPTER 5
Workload Analysis and Memory Scalability

In this chapter we explore the effects of strong scaling on workload efficiency. Using measured HWM and runtime data, for a cross section of scientific applications, we explore how strong scaling can be used to reduce application memory HWM.

With this information we simulate, using the Maui scheduler, the total runtime for a workload of jobs on different machine configurations with decreasing memory-per-core ratios. By reducing the available memory-per-core, we force the jobs to be run at increasingly large scale, where there is a naturally lower level of parallel efficiency, and thus increase overall runtime.

Applications with better memory scaling, or inherently lower memory consumption, are more versatile when it comes to runtime configurations in low memory environments, thus minimising the effects of memory-per-core reductions. We use this behaviour to drive an investigation into two techniques to improve memory scalability, with a specific interest in reducing ghost cells. We demonstrate how processor decomposition choice and on-node parallelism can drastically improve memory scalability, demonstrated through a benchmark application (SNAP).

During the first portion of this chapter (Sections 5.2 and 5.3), where we simulate workflow scalability, we utilise memP to obtain HWM data for the applications. Although this data collection occurred prior to the development of WMTools, we would expect the same results had WMTools been utilised. In the latter portion (Section 5.4), where we analyse the memory scalability of SNAP, we utilise WMTools as it has support for openMP which is integral to the analysis.

The decision to use a selection of benchmark applications for the first portion of this chapter is to try and capture the varying memory scalability of different scientific methods, and implementations. They are used to demonstrate the methodology of analysing workflow memory scalability, rather than to represent the exact magnitude of effects.

Similarly we utilise the SNAP benchmark due to its ability to handle different processor decomposition methods, and its internal hybrid parallelism. What we present is the methodology of memory analysis, using SNAP as the medium, although the behavioural results are applicable for a wide range of applications, although the exact results will be application dependent.

The workload efficiency analysis presented in this chapter was first published in [105].

5.1 Related Work

Maui has been used to simulate various aspects of HPC execution, including predicting job start times [72], scheduling policies [59] and resource partitioning [116].

These works often take known workloads, measured from a system, and try to optimise execution by simulating different scheduling configurations. Our work differs as we are generating artificial workloads, with a fixed scheduling policy, where the workload is varied based on job size to satisfy differing memory requirements.

One key principle we have employed is the avoidance of job dependencies, which are often found in workflows, as this would complicate the ordering of execution. Instead we allow jobs to be scheduled in any order and enable scheduler backfilling where appropriate.

In the latter half of this chapter we explore some memory saving techniques, designed to improve scalability, which can be adapted at runtime. A good analysis of the available memory savings afforded by hybrid parallelism (in this case MPI and OpenMP) for a large scale scientific framework is presented by Meng *et al.* for Uintah [81]. They document how memory constraints are becoming increasingly problematic at very large scale when utilising flat MPI, and present an investigation into hybrid MPI and OpenMP parallelism. By evaluating the memory savings from both a ghost cell and global mesh data perspective, they are able to build an algorithmic memory consumption model from which they conclude they can save up to 90% of memory over flat MPI. Whilst this model does not factor in some runtime and environmental memory consumption, such as communication buffers, it does indicate the potential memory savings available.

This modelling does not factor in the use of additional memory, for performance aspects such as OpenMP 'private' data, where there is replication to avoid race conditions or the need for memory locks. Such performance techniques will obviously increase the memory consumption in the practical case.

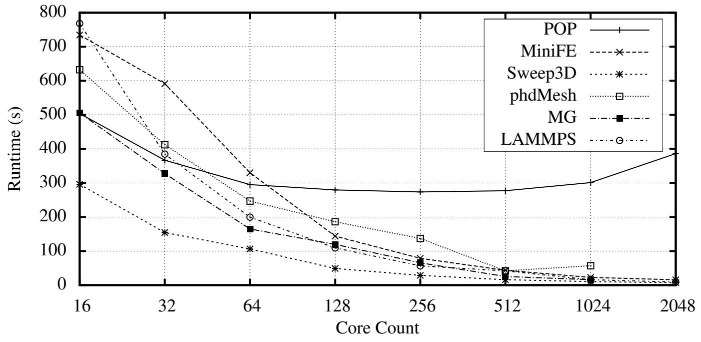
In our research we instead measure the actual savings achieved by a hybrid parallel code, and at the same time measure the performance impact. We employ aspects of this modelling technique in Chapter 7, where we utilise a memory model to predict savings from the use of OpenMP based node level parallelism.

5.2 Application Memory Consumption

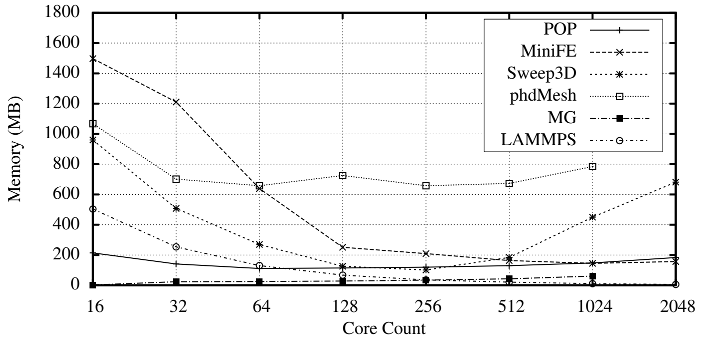
To simulate a workload we take a selection of six applications, representing a cross-section of scientific domains: POP, miniFE, Sweep3D, phdMesh, MG and LAMMPS. These applications all exhibit different behaviour, allowing us to accurately simulate a mixed science workload.

For this section we use memory data collected using memP, as this research was conducted before the development of WMTools. As WMTools has been shown to produce results consistent with memP we would anticipate this study to draw the same conclusions had WMTools been utilised for data collection.

Firstly we benchmark these applications for their strong scaling behaviour of



(a) Benchmark applications runtimes



(b) Benchmark applications HWM

Figure 5.1: Runtime and HWM scaling behaviour for benchmark applications on Hera

the LLNL Hera platform, recording the runtime and HWM at each core count, as shown in Figure 5.1. From Figure 5.1(a) we can see that the majority of the applications scale reasonably well in terms of parallel efficiency. We do see some subtly different scaling behaviour, where applications such as POP do not scale efficiently at high core counts.

Figure 5.1(b) illustrates the memory consumption profiles of these same applications. To a certain extent we see a similar behaviour in memory as we do in runtime. Many applications scale very well, but some applications begin to use more memory on the higher core counts. Our choice of mixed science applications establishes a wide spread of memory consumption values, indicative of a real world mixed science workload.

5.3 Simulating Effects of System Memory Loss

In this section we investigate the impact on workload performance, resulting from execution on platforms with reduced memory-per-core ratios. With lower memory-per-core ratios it becomes impossible to run large memory jobs on small core counts, thus strong scaling is required to reduce the per-core memory. The effect of strong scaling an application usually results in a reduction of parallel efficiency, unless super linear scaling occurs [99]. Whilst the individual job may complete in a faster time, the resource usage will negatively impact on the completion of a workload.

We simulate this occurrence through the use of artificial workloads made from our six benchmark applications, simulated on machines with ever decreasing memory per core values.

5.3.1 Workload Construction

We generate three different workloads with different application weightings, to avoid bias towards any particular application. We note that from Figure 5.1(b) that the application phdMesh, for the current problem size, does not scale its memory consumption below 600 MB per core. For this reason we construct one of our workloads to exclude phdMesh, allowing us to simulate scaling to machines with less than 600 MB per core.

	Workload 1	Workload 2	Workload 3
POP	40%	20%	15%
miniFE	10%	30%	10%
Sweep3D	15%	5%	35%
phdMesh	15%	10%	0%
MG	10%	20%	25%
LAMMPS	10%	15%	15%

Table 5.1: Mixed application workload compositions

Table 5.1 details the percentage contribution of each application for our three workloads.

5.3.2 Machine Simulation

To simulate workload execution we utilise the Maui scheduler in simulation mode. The scheduler simulator is is designed to allow users to safely evaluate arbitrary configurations, but in this case it allows us to simulate a production environment with various different memory restrictions.

During the simulation each job is submitted in such a way that its runtime requirements satisfy an artificially imposed limit on the amount of memory available per core. We initially set our memory-per-core at 1.5 GB, as this corresponds to the maximum usage seen in Figure 5.1(b), and then reduce this to 1280 MB, 1024 MB, 768 MB, 682 MB, 512 MB and 256 MB per core. As Hera has 16 cores per node, as detailed in the appendix in Figure A.2, these memory-per-core figures represent 24 GB, 20 GB, 16 GB, 12 GB, 10.7 GB, 8 GB and 4 GB per node. The choice of 682 MB per-core was utilised to represent the use of a hex-core processor with 4 GB, which whilst not applicable to Hera is interesting for other platforms.

We use 2048 cores of Hera as a basis for our simulated machine (defined using a Maui resource trace file), and our simulated workloads consist of 1000 jobs in the proportions defined in Table 5.1 (defined by a Maui workload trace file). We maintain the existing polling system in Maui and replicate normal use by specifying wall-times in excess of known execution time, and allow the scheduler to backfill jobs where possible. Since the simulator does not allow us to submit additional jobs while the simulation is running, we must include the complete job list at the start and rely on polling to ensure a stream of jobs to the system. Due to the impact of job order we simulate each different workload with ten random orderings (repeated on all skews), and record the average runtime.

Skew Factor

Even when applications are memory constrained they can still be run on various core counts. Typical users will often not select core counts for their jobs optimally. Many jobs will be run at high core counts, therefore lower parallel

	Skew 0.25	Skew 0.5	Skew 0.75
Workload 1	17.94%	13.21%	9.78%
Workload 2	27.76%	18.32%	12.67%
Workload 3	12.36%	8.09%	4.19%

Table 5.2: Percentage runtime increase from 1536 MB to 682 MB per core

efficiency, to reduce the overall job runtime, without consideration to global workload efficiency.

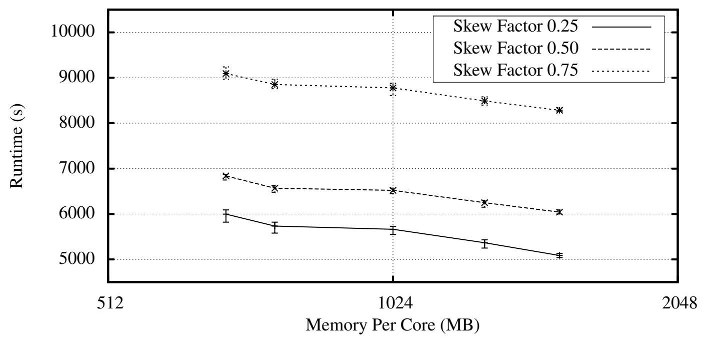
We therefore base the selection of application core count in our workloads on a partitioned Gaussian distribution. The distribution determines how probable it is to select a particular core count, a skew factor is introduced to adjust this distribution. A skew of 0.25 will ensure that the distribution has a bias towards core counts that are closer to the minimum possible core count configuration. A skew of 0.75 will generate a more even distribution between possible core count configurations, to represent users who prioritise the turnaround time of their jobs, rather than workload efficiency. In this work we are not evaluating the efficiency of scheduling policy, but rather trying to account for the variation in user preference. As a result, we will focus on the general trend of behaviour rather than specific values.

5.3.3 Performance Analysis

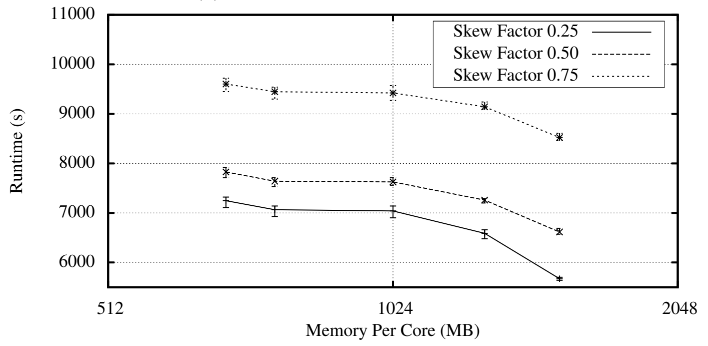
We compare the time-to-completion for three different mixed-science workloads as the available memory per core is reduced. Results are shown with three different core-count selection skew factors (0.25, 0.5 and 0.75) with an identical Maui scheduler simulator configuration used throughout.

The results for the three different workloads can be found in Figure 5.2. For each, the average workload completion time is plotted as a trend line with maximum and minimum times from the different job orderings reported as error bars. Several observations can be made regarding these results.

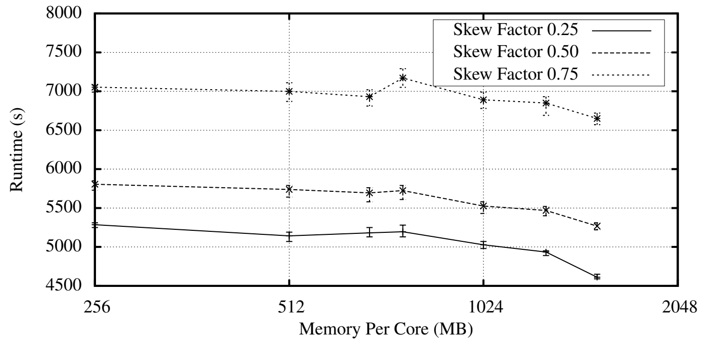
Firstly we can observe a trend supporting our initial hypothesis, that de-



(a) Workload 1 simulated runtime



(b) Workload 2 simulated runtime



(c) Workload 3 simulated runtime

Figure 5.2: Simulated runtimes for workloads with different memory restrictions

creasing memory-per-core ratios would adversely effect workload efficiency. In all cases a reduction in memory results in an overall increase in workload runtime.

Secondly we can see that in all cases the use of a lower skew factor, representing job submission at the lowest viable core count, results in a more efficient runtime configuration than executions with a larger skew factor. Although these low skew configurations are more adversely affected by the decreases in available memory per core, as a factor of tighter job scheduling. A low skew job is more likely to be run at larger scale when less memory is available, than a high skew job which may have already been at a large core count. Our results exhibit an average runtime increase of 19.35% for a skew of 0.25 as opposed to an increase of only 8.88% for a skew of 0.75, when the memory-per-core is reduced from 1.5 GB to 682 MB.

Thirdly we show that the magnitude of runtime increase is workload dependent, based on the composition of highly scalable applications. We see that on a whole Workload 3 is much more amenable to memory reductions, when compared with the other workloads, as visible in Table 5.2. We also see that within workloads the job ordering has minimal effects, and does not alter the general trends.

This observation emphasises the importance of low application memory consumption and good memory scalability. Where applications have poor scalability, or naturally high consumption, decreases in memory-per-core will force execution on ever increasing core counts, adversely effecting workload efficiency. Thus good scalability makes applications more versatile in terms of core count configuration, especially in low memory environments.

5.4 Understanding Scalability

In Section 5.2 we presented the memory profiles of a number of different applications. What is clear from these results is that there is no fixed trend of memory scaling for different applications.

One might naïvely assume that when strong scaling the memory consumption will halve with every successive doubling of core count. In a similar vein one might expect memory consumption to stay constant through weak scaling. Unfortunately, much like runtime, neither of these two expected trends are realistic.

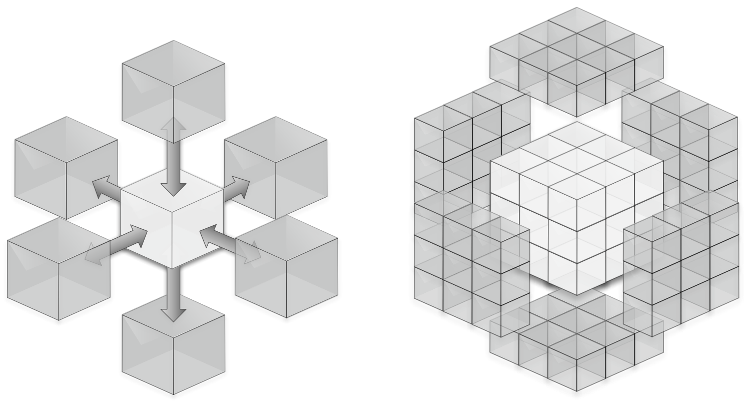

(a) Seven-point data stencil (b) Ghost cells for 3^3 cube

Figure 5.3: Data dependency and ghost cells in a cube

Complexities within the application mean that there are some allocations which are always required, generating a level of constant consumption, regardless of scale. Additionally there are some allocations which will actually grow in size with core count.

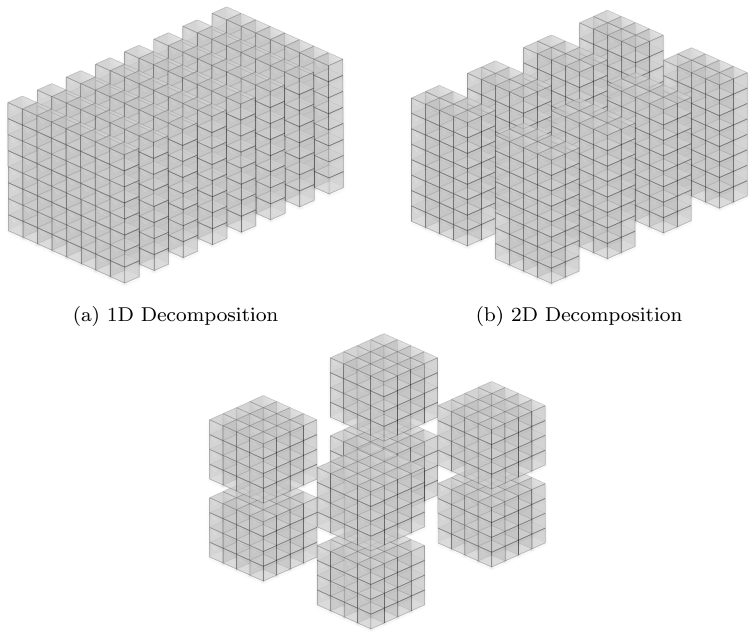
In this section we discuss the role of ghost cells in memory scalability, and methods of reducing their impact. We also briefly discuss the role of communication buffers in memory consumption – a topic we expand upon in Chapter 6.

We explore the real life memory savings achievable from the techniques presented above, through the use of the LANL benchmark SNAP, on the LLNL platform Cab. Using SNAP we test the memory and performance differences between a 1D and 2D decomposition and the use of hybrid parallelism.

5.4.1 Ghost Cells

The choice of decomposition strategy can make a marked difference on ghost cell numbers. The ratio of ghost cells to data cells will in turn affect the amount memory available to store local domain data.

To demonstrate the influence of decomposition let us consider a cubic mesh. In this example we assume the algorithm operates a simple seven-point stencil,



(a) 1D Decomposition (b) 2D Decomposition

(c) 3D Decomposition

Figure 5.4: Decompositions of an 8^3 cube onto eight processes

meaning each cell requires data from the six cells surrounding it, as illustrated in Figure 5.3(a). Such a data dependency requires each processor to maintain a 'halo' of ghost cells, demonstrated in Figure 5.3(b). On boundary faces this halo data may be available locally, but on internal faces this data must be communicated from other processors, thus also represent communication. Ghost cells are stored locally, but rarely computed locally, and thus are a duplication of data that takes up additional space to aid performance.

5.4.2 Processor Decomposition

Figure 5.4 demonstrates the different methods of decomposing an 8^3 matrix onto 8 processors. The options presented are 1D, 2D and 3D, all giving an even decomposition of 64 cells per processor. Thus, in terms of our seven-point stencil algorithm, these three decompositions are numerically equal in terms of workload and memory.

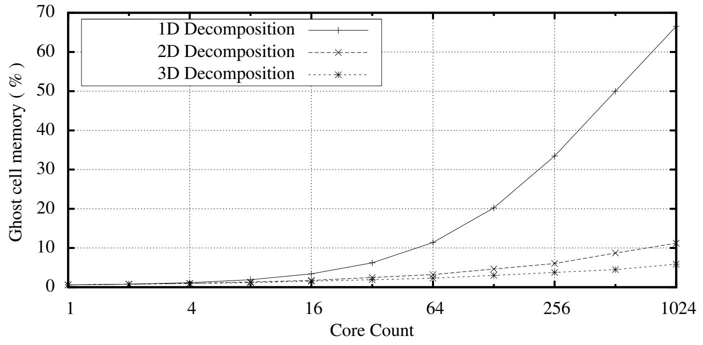


Figure 5.5: Proportion of ghost cells as a percentage of total cells in different decompositions of a 1024^3 cube

In this section we look at the memory effects of the ghost cells required to support each decomposition. Specifically we investigate the decomposition of a 1024^3 cube. For each processor count we compute the best decomposition, and calculate the number of ghost cells required, as a percentage of the local memory.

Figure 5.5 represented the percentage of the total cells of the decomposition which form the halo data. It is clear to see the influence of the decomposition on this ratio of ghost cells to data cells. As all decompositions are equal and equivalent the only only difference at each processor count is the nature of the decomposition.

We see that the 3D decomposition has a better scaling than the 2D decomposition and significantly better than the 1D, improving the memory scalability significantly.

We note that these measurements do not represent any analysis on the suitability of such decompositions on the underlying algorithm, or their communication costs. They only represent the volume of memory required to store data which is not computed, and can be configured.

The reason the 3D decomposition outperforms the other decompositions is it tries to configure the shape of the local problem to closely match the global problem shape. As the global problem is a cube, it is most efficient for that to represented in the local problem shape. What we are thus representing is the ratio of surface area to volume, for the local problems. The lower this value the lower the ‘wasted’ memory on each processor.

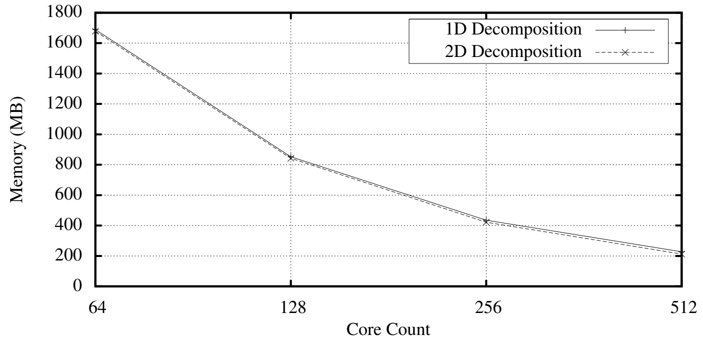
Is this experiment we have focused on a 7-point stencil; as the data dependency grows to 13- and 25-point stencils the problem is amplified. In such cases the halo region would usually be two and three cells deep, respectively.

Empirical Study

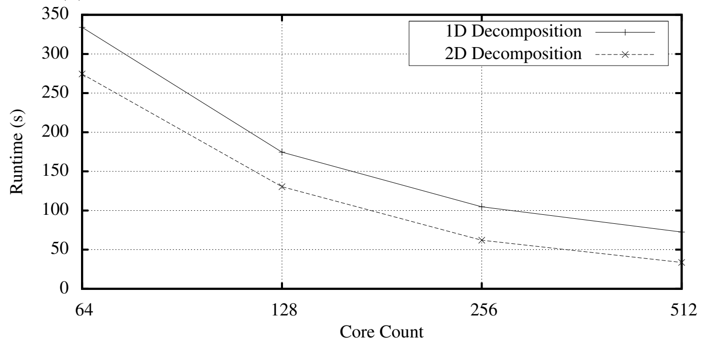
Although SNAP can operate on a 3D data domain, it only supports 2D processor decompositions, due to the inefficiencies of 3D decompositions of sweep based applications [6]. Thus we are only able to test the memory savings between a 1D and 2D decomposition. Figure 5.5 illustrates the potential differences between the two decompositions, when a single cell halo is required. To demonstrate this effect we used a large data mesh of 512^3cells, with four energy groups and 4 angles. This configuration was designed to have a large and easily decomposable mesh, whilst minimising the memory requirements from other parameters.

From Figure 5.6 we can see both the memory consumption and runtime impact from the choice of a 1D and a 2D decomposition. Figure 5.6(a) illustrates that we see a memory reduction of 6%, on 512 cores, from the 2D decomposition over the 1D. We also note, from Figure 5.6(b), that there is is marked performance improvement for the 2D decomposition, as it is naturally more efficient in terms of communication.

Whilst the memory savings presented in this section may not be of the same magnitude of those proposed in Section 5.4.2 our memory reduction is an absolute reduction, thus factoring in all sources of memory consumption not just problem cells. The suggested savings are based on calculations of ghost cell ratios to data cells, and not all of the data will contain ghost cells, thus will not exhibit a reduction with the change of decomposition.



(a) Decomposition comparison memory consumption for SNAP



(b) Decomposition comparison runtime for SNAP

Figure 5.6: 1D and 2D decomposition comparison for SNAP

5.4.3 On-node Parallelism

Data dependencies can have a serious impact on design decisions, at many different layers of the development process. The choice of processor decomposition is one area where memory utilisation can be tuned with respect to the storage of ghost cells.

An alternative solution would be to avoid storage of the ghost cells altogether. With the rise of multicore processors the density of processors-per-node has increased greatly in the past few years. With respect to processor decompositions this means we are storing a buffer of a neighbouring thread's cells, which may actually reside in the same memory space. These shared memory spaces may constitute Non-Uniform Memory Access (NUMA) regions, which allow for fast access of data from multiple cores within the same processor.

For such a scenario, programming models such as hybrid MPI and OpenMP may be more appropriate. For a hybrid implementation the ghost cells would only need to be stored for the bounding region for the whole processor, rather than each individual core. Again this minimises the surface to volume ratio, on a per processor basis. Depending on the number of cores per processor, this could have a dramatic effect on the number of ghost cells stored.

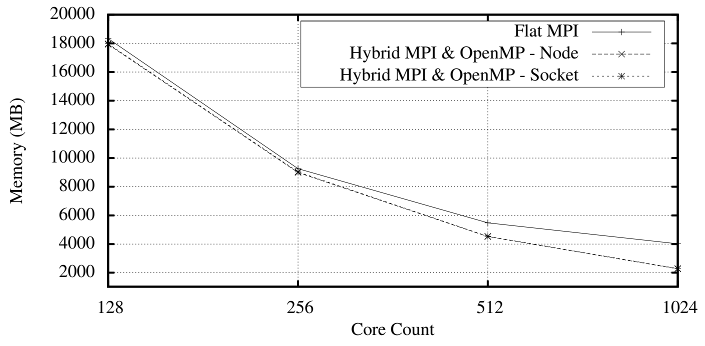
An extension of this idea of replication avoidance would be to remove it altogether, not just at the processor level. Through the use of PGAS languages (or one-sided communications in languages such as MPI) it would be possible to engineer a solution which communicates cells individually rather than store them – though practically some form of buffering would be desirable from a performance perspective. The viability of such an implementation would be based on the increased communication cost of these transfers and would present a direct tradeoff between memory consumption and runtime.

Empirical Study

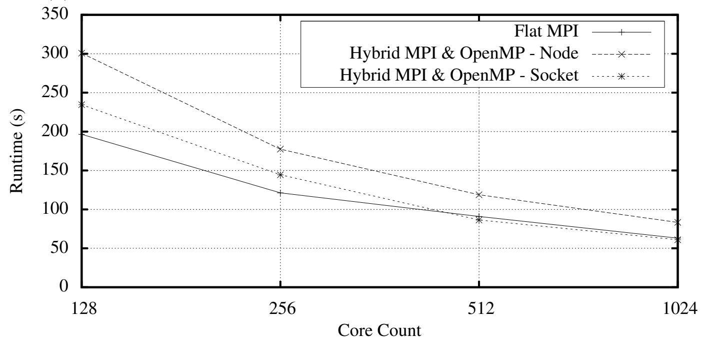
To study the effects of using hybrid parallelism, in the form of MPI and OpenMP we compare the memory consumption and performance of SNAP. We compare flat MPI against hybrid MPI and OpenMP run in both node level (16 OMP threads) and NUMA level (8 OpenMP threads) per MPI task. To demonstrate these effects on SNAP we used a 2D processor decomposition and a smaller global mesh (96^3) but with a much higher number of energy groups and angles: 40 and 500 respectively.

Figure 5.7 presents a memory and runtime comparison for the two different parallelism models available: flat MPI and hybrid MPI and OpenMP. What we can clearly see is that there is a significant memory saving from the use of OpenMP, shown in Figure 5.7(a); a 43.5% reduction at 1024 cores. As with the decomposition comparison, the memory reduction is proportional to the core count, due to the increased ratio of ghost cells to data cells.

In the case of the hybrid parallelism model we do see a performance degradation over the flat MPI configuration, shown in Figure 5.7(b); a 32.0% increase



(a) Parallelism model memory consumption comparison for SNAP



(b) Parallelism model runtime comparison for SNAP

Figure 5.7: Flat MPI and hybrid MPI and OpenMP comparison for SNAP

at 1024 cores.

In both cases we also compare the implication of running with a single MPI task per node against one MPI task-per-socket. This is because a socket represents a distinct NUMA region, and so memory accesses are constrained to the memory attached to the specific socket. A more detailed representation of the memory structure of Cab is presented in the appendix in Figure A.1.

What we observe is a marginally smaller memory reduction to that of the single MPI task per node hybrid model, as a result of the introduction of one internal boundary of ghost cells. This is still a significant reduction when compared to the flat MPI configuration, a 42.9% reduction at 1024 cores.

In addition we observe an increase in performance, over the task per node model, approaching and even surpassing the performance of the flat MPI configuration, showing a 3.3% decrease in runtime at 1024 cores.

From the comparison of these three configurations we begin to understand the available memory-runtime tradeoffs afforded by hybrid parallelism models. These results were based on eight core processors - as the cores-per-node ratio increases with future technology, the memory savings from multiple levels of parallelism will increase, helping to address the issues of decreasing memory-per-core ratios.

5.4.4 Communication Buffers

The topics previously covered in this section have been aimed at improving scalability where strong scaling results in monotonically decreasing memory consumption. Sadly this is not the case, as we saw at the start of this chapter in Figure 5.1(b), sometimes memory consumption can increase at large scale. One of the factors behind this increasing component is the usage of communication buffers, either explicitly within the application or within the MPI implementation. The concept of reserving a block of memory proportional in size to the number of ranks in the job will clearly result in increases in memory consumption at large scale.

In Chapter 6 we present a study on these effects, identifying and analysing the memory reserved by MPI for receiving communications from other processes.

5.5 Summary

In this chapter we have demonstrated how gaining knowledge about an application's HWM, from tools such as WMTools and memP, allows us to analyse the effect of strong scaling on memory consumption.

By using a selection of scientific applications we illustrate how strong scaling can be used to accommodate reductions in memory-per-core. Utilising higher core counts to save memory has a negative impact on the application's parallel efficiency, and we use this information to model the overall runtime increase incurred by memory-per-core reductions. Using the Maui scheduler we simulated the execution of three different workloads, composed of different job mixes, to study these effects.

We illustrated how a reduction in memory-per-core from 1.5 GB to 682 MB results in a workload runtime increase of 13.8%. We demonstrated the benefits of improving memory scalability by constructing a specific workload of memory scalable applications, and demonstrate a runtime increase of only 10.2% when memory-per-core is reduced from 1.5 GB to 256 MB.

We continued this chapter by with a look at some of the causes of poor memory scalability, with specific focus on ghost cells, and the way processor decompositions can effect their dominance. Using WMTools we were able to analyse the variation in HWM afforded by different processor decompositions (1D and 2D) on the SNAP benchmark – observing a 6% reduction in HWM at 512 cores. Similarly we analyse different parallelism modes within SNAP, presenting a comparison of flat MPI with hybrid OpenMP and MPI. From this we were able to demonstrate a saving of nearly 43% with an accompanied 3% decrease in runtime, with SNAP at 1024 cores running one MPI task per NUMA region.

Using WMTools to analyse the memory consumption at node-level, rather than the per-core level available in other tools, enabled a fair comparison of these parallelism techniques.

CHAPTER 6
MPI Memory Consumption

In this chapter we address the memory consumption of the MPI library at increased scale. Specifically we investigate a known problem of poor memory utilisation on InfiniBand network hardware [66, 67, 75, 115].

The problem stems from the necessity, within current implementations, to store certain information for each communicating pair of nodes in a job. Thus as the core count of the job increases the memory requirements of these communication buffers scale accordingly.

Firstly, we present an analysis of the problem, as exhibited by the OpenMPI MPI implementation on two different InfiniBand implementations: QLogic and Mellanox. This analysis is performed using WMTools, to identify the contribution to memory of the MPI library at time of HWM.

Additionally, we present an investigation into available solutions, including runtime configurations and vendor-provided libraries, and evaluate their impact on both memory and application runtime.

We use WMTools in the context of Orthrus, a generic 3D implicit linear solver benchmark developed at AWE, to analyse this MPI memory consumption on specific InfiniBand implementations. Our analysis utilises two machines with InfiniBand from different vendors, QLogic on Cab and Mellanox on Kay, to understand the key fundamental differences in MPI memory consumption. The similarities in the platforms, as expressed in the system diagrams in Appendix A.1, allows us to compare results with a high degree of confidence.

Using Orthrus to drive the PETSc solver library [8] using the `Block JACOBI` preconditioner and `GMRES` solver, we solve a 50^3 per-core weak scaled problem. On both platforms we utilised the Intel 12 compiler and OpenMPI 1.6.3, to build and run Orthrus.

Our reasoning for using the Orthrus benchmark in this chapter is down to the internal communication structure. The dependence on point-to-point communications, and the associated scaling of this communication pattern makes it the perfect candidate to examine MPI memory consumption. Due to these characteristics Orthrus had previously illustrated memory scalability issues, and thus represented a good candidate for memory analysis. Whilst the artefacts presented in this chapter are exposed through Orthrus they exist in all codes, with varying magnitudes. As such any memory savings presented here will be proportional to the initial artefact and will be code dependent.

We use this benchmark application to definitively demonstrate the memory scaling issues of MPI under certain conditions, allowing us to identify hardware and software configurations of specific interest. We examine the effectiveness of runtime configurations, where communication buffer sizes are constrained, in reducing MPI memory consumption. Lastly, we investigate the use of vendor-specific communication libraries, allowing the optimisation of communication protocols for specific hardware.

The MPI consumption analysis research presented in this chapter was published in [102].

6.1 InfiniBand Communication

InfiniBand supports five modes of transport: Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC), Unreliable Datagram (UD) and Raw Datagram. RC is the most common strategy amongst MPI implementations, due to its support for Remote Direct Memory Access (RDMA) and so enhanced performance. RC and UC require a connection to be made between every queue pair (QP), and memory allocated in the event of communication, an inherently non-scalable method. RD is similar to RC but is designed to be inherently more scalable – only one QP is used to communicate with other RD

QPs.

UC and UD differ from RC and RD as they do not provide acknowledgements for messages, and therefore are often impractical for MPI network connections.

Raw Datagram provides the facility to communicate messages which are not interpreted.

Messages sent between QPs are tracked by a send Work Queue Entry (WQE), thus the number of WQEs allotted per QP defines the maximum number of outstanding send-receive operations.

6.1.1 MPI Receive Queues

There are three different types of receive queue in MPI: Per-Peer (P), Shared Receive Queue (SRQ) and eXtended Reliability Connection (XRC). Per-peer receive queues allocate dedicated buffers to each sender thus this memory consumption will grow with the number of communicating ranks in the job. SRQs partition a buffer space which can be utilised for messages from any source, without dedicating space to any particular source. Thus SRQs allow both WQEs and buffers to be reused, rather than locked to a single QP. Additionally there is the eXtended Reliability Connection (XRC) receive queue, which is specifically designed for Mellanox hardware to reduce the number of QPs required, as it allows a single receive QP to be shared between multiple SRQs [114].

6.1.2 InfiniBand Interface

The communication from application level to InfiniBand hardware is via a user-level API - the widely adopted OpenFabrics Enterprise Distribution (OFED) industry standard. OFED provides a programming interface for libraries, such as MPI, to enable RDMA and kernel bypass [97].

Vendors are then able to provide customised libraries built on OFED, optimised for their hardware. Similarly vendors can provide optimisation libraries for their OFED distribution to provide enhanced features.

Mellanox MXM

The Mellanox optimisation library MellanoX Messaging (MXM) was specifically designed to address some of the issues their hardware faces with resource utilisation at scale [111]. A transition to the UD communication model is designed to improve point-to-point memory consumption, by alleviating the need for QP-specific receive queues. The MXM library is specifically targeted at scalability and is not intended to improve the performance of point-to-point messages, or address collectives in anyway. Collective optimisations are provided through separate Mellanox technologies: Fabric Collective Acceleration (FCA) for software-based optimisations, and CORE-direct for hardware-based optimisations. These optimisations are not the focus of this study, due to their minimal memory overheads.

QLogic PSM

The QLogic Performance Scaled Messaging (PSM) library is an alternative to InfiniBand verbs (the traditional interface to InfiniBand) designed to increase performance [107]. PSM is specifically designed for HPC message passing requirements, and is optimised for QLogic's on-load approach to communication, where work is delegated to the host CPU. As such, PSM provides improvements to both point-to-point and collective communications.

6.2 Related Work

The scalability of MPI has been a cause for concern for many years, both in terms of runtime and resource consumption [7, 115].

As early as 2004, Liu *et al.* demonstrated memory issues with MPI over InfiniBand [75]. Their experiments, comparing the memory scalability of MPI on different network fabrics, demonstrated a problem with the use of the RC service. The memory consumption of InfiniBand is shown to be significantly worse than both Quadrics and Myrinet, with a scaling proportional to core count.

In [66], Koop *et al.* discuss the memory requirements of connections tested through the MVAPICH MPI implementation. They determine the additional memory required per connection with different numbers of WQEs allocated per QP, which ranges from 8.8 KB, for 5 WQEs-per-connection, to 132.8 KB for 200 WQEs per connection (the default in MVAPICH). They demonstrate a reduction in memory consumption from 1 GB-per-core at eight thousand cores using the default 200 WQEs-per-QP, to less than 90 MB-per-core when using 5 WQEs-per-QP, representing an $11.3\times$ decrease. As this reduction in WQEs will directly affect the messaging rate (only significant for small messages), they also investigate the use of message coalescing to improve small message performance at low WQE counts.

In a subsequent paper, Koop *et al.* document an implementation of MVAPICH based on UD rather than RC [67]. They present findings on the memory consumption comparing two different SRQ configurations and a UD implementation for a fully connected send-receive run, scaled to sixteen thousand cores. They show an 80% reduction in memory consumption at sixteen thousand cores, when comparing the UD and unoptimised SRQ-based RC configuration. These results also demonstrate near-flat memory consumption scaling, ensuring the viability of this technique at very large scale, whilst achieving comparable performance and reliability.

The memory implications of scaling have been evaluated for other network fabrics, and MPI implementations, to enable MPI on low memory architectures such as the Blue Gene/L [37].

Shida *et al.* released a report detailing some of the issues they faced porting OpenMPI to the K computer at RIKEN in Japan, comprised of 88,128 nodes with 2 GB main memory per-core[112]. A large focus of this porting effort was effective MPI memory management through the use of runtime profiles, which control communication buffer size. We demonstrate a similar technique in this thesis, with the use of the BullXMPI low memory footprint profile.

In [113], Shipman *et al.* extended the idea of the shared receive queue to allow resource pooling, through the use of buckets. The idea behind bucketed-SRQs (B-SRQ) is that by having a number of receive buffers of different sizes, buffer utilisation can be optimised for communication. With this receive queue optimisation they are able to demonstrate efficient utilisation, resulting in an increase in overall performance for a large selection of applications.

Other research has looked specifically at the memory registration process of Mellanox hardware, but traditionally the focus has been on performance rather than consumption [86].

Whilst this prior research has identified the issue of poor memory scalability under certain conditions, they have not studied the effects on a real application. Where applications have been used they have been communication benchmarks, which are not representative of normal application behaviour. Additionally this prior research has not detailed their memory analysis methodologies, or been clear on how the MPI consumption is measured.

Where solutions have been proposed in literature, they have again failed to demonstrate the effects on a real application. Such an approach masks the true implications of the solution with respect to real users.

In this chapter we clearly demonstrate the issue with a real application, and illustrate why it is felt with such gravity, and decompose the effects of the available solutions using our in-depth analysis tools.

6.3 Application Profile

In this section we perform an application analysis, on the Orthrus benchmark, to gain an understanding into communication and memory consumption behaviour. By looking at the communication patterns of the application we can understand how each technique should reduce memory consumption and impact performance. By measuring the number of sources of communication for each process we can understand the ideal number of receive queues, and the size of these communications we can understand the required size of these queues. Further, we can speculate about the performance of a queue configuration based on the balance of these two attributes.

As receive queue count and size are the primary cause for MPI memory consumption this information can be used to design low memory configurations with minimal performance implications.

6.3.1 Application Communication Classification

To gain understanding of Orthrus we first perform an analysis of its communication profile. We evaluate the ratio of point-to-point to collective communications, and measure the source and destination of point-to-point communications to understand the communication structure. Lastly, we analyse the message sizes used in the point-to-point communications to help understand the effect of receive queue size.

Point-to-Point and Collectives Analysis

Table 6.1 represents the breakdown of total MPI communication calls made during execution on 64, 128 and 256 cores. From this table it is clear to see that the dominant communication type, in terms of frequency, is point-to-point. We also note that the growth factor between 128 and 256 cores suggests that the frequency of point-to-point messages (3.6×) is increasing at a faster rate than that of collectives (2.5×). These results suggests that optimisations targeted towards point-to-point communications is likely to of considerable benefit.

Source - Destination Analysis

In Figure 6.1 we illustrate the communication pattern of our benchmark application run on 128 cores. During an instrumented execution, we measure the source and destination of every point-to-point communication. The colour of a cell indicates the density of communication between the source and destination processes, where black represents maximum observed communication through

Library Call	Type	64 Cores	128 Cores	256 Cores
MPI_ISend	Point-to-Point	118,416	446,576	1,618,632
MPI_IRecv	Point-to-Point	118,416	446,576	1,618,632
MPI_Gather	Collective	704	1,408	2,816
MPI_Gatherv	Collective	512	1,024	2,048
MPI_Allgather	Collective	3,328	6,656	13,312
MPI_Allreduce	Collective	105,088	263,168	652,288

Table 6.1: Different communications at 64, 128 and 256 cores

a grey-scale to white representing no communication. From this data we can see the complexity of the problem decomposition and associated processor layout; in this case each processor sends data to $\approx \frac{1}{4}$ of the other processors and receives data from another $\approx \frac{1}{4}$ of the processors. When visualised as a 3D processor decomposition each processes sends data to all processes in a plane of the x-dimension and receives data from all processes in a plane of the z-dimension. This behaviour is not representative of the communication profile of the underlying seven-point stencil algorithm, and may indicate issues with the communication structure.

The fact that each processor also receives from a large number of other processors means that this application will require a large number of QPs. This suggests that the application is a prime example to demonstrate MPI memory scalability problems, and to evaluate potential solutions.

Point-to-Point Message Size

Figure 6.2 augments Figure 6.1 by looking at the size of these point-to-point messages. From this figure we can see three very distinct clusterings: the first in the 0 B to 4 B range, the second spanning from 16 KB to 64 KB, the last group spanning from 128 KB to 512 KB.

What we ascertain from Figure 6.2 is that to optimise the receive queues for Orthrus, we are required to support a small percentage (≈20%) of very small messages and a large percentage (≈80%) of larger messages. There is little to be gained from a large number of medium sized receive queue buffers, and so

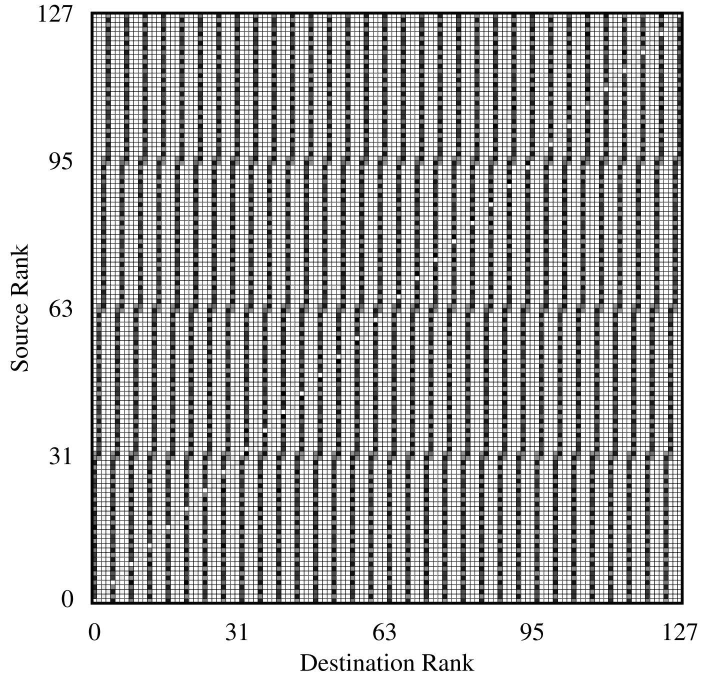


Figure 6.1: Source-destination distribution for point-to-point messages on 128 cores

large buffers should be prioritised.

6.3.2 Application Memory Profile

We now perform some memory analysis to establish a memory consumption profile, which will help us understand the importance of memory reductions. By understanding how much memory MPI consumes, and where in the application the consumption occurs, we are able to gauge the potential savings from our different solutions.

Temporal Trace

Figure 6.3 shows the temporal memory profile of Orthrus on 128 cores. We can see that Orthrus has a very regular profile, with a repeating pattern covering the 20 time-steps. We note that there is a 'ramp-up' phase during the first

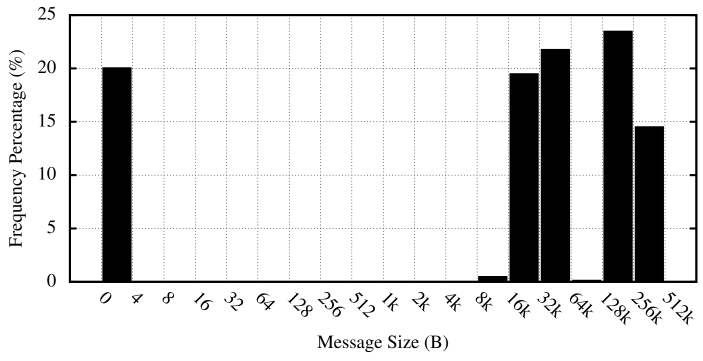


Figure 6.2: Frequency distribution of message sizes on 128 cores

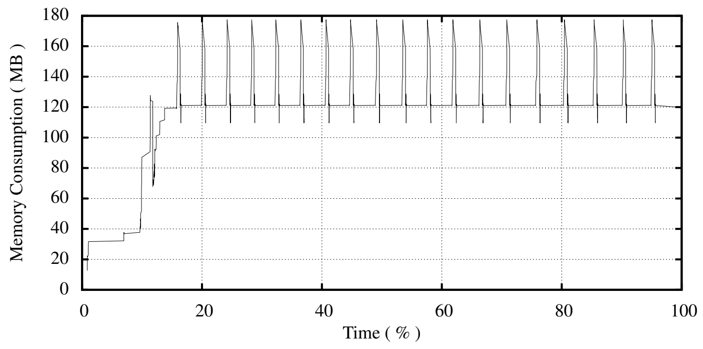


Figure 6.3: Temporal memory usage for Orthrus on128 cores

time-step where allocations are made.

We can see that the height of the spikes is consistent and non-increasing, indicating well-managed memory and suggests there are no memory leaks.

6.4 Identifying MPI Memory Consumption

Figure 6.3 showes us that the memory profile of Orthrus is fairly regular, but does not reveal the source of consumption. To look for MPI memory consumption we start at large scale, where the effects of MPI artefacts will be greatest. If we study the functional breakdown at the point of HWM on a 1024 core execution, we can observe where the memory consumption is occurring, and identify MPI consumption.

```
# HWM Functions file from WMTools - WMTrace/trace -0.functions
#        HWM of 249748421(B)
#
# MPI Memory summary: 92563941(B)
#        (37.0629%) of memory attributed to MPI (libmpi)
#
# High Water Mark Function Breakdown

Call Stack: 100 Allocated 67133640(B)
        (26.8805(%) ) from 8015 allocations

libmpi.so.1
-mca_btl_openib.so
 ..............
------------openmpi/mca_coll_sync.so
-------------libmpi.so.1
--------------PMTM_parameter_output
 ..............
-------------------main
--------------------/lib64/libc.so.6
---------------------_start
```

Listing 6.1: Orthrus HWM functional breakdown on 1024 cores

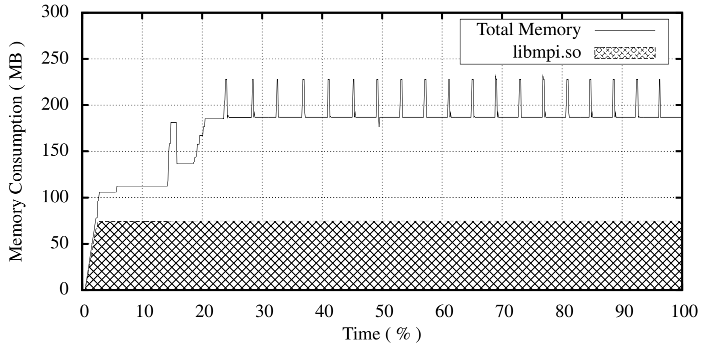
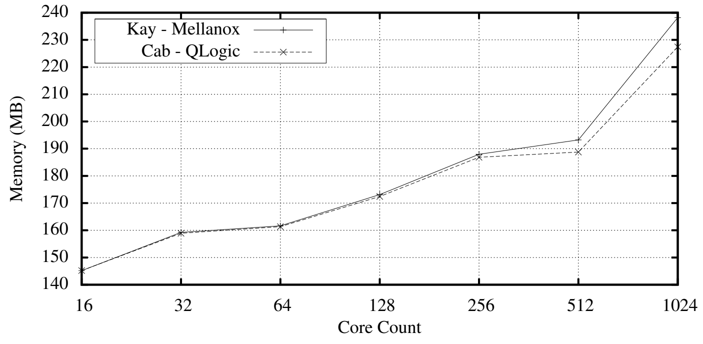


Figure 6.4: Orthrus MPI memory consumption at 1024 cores

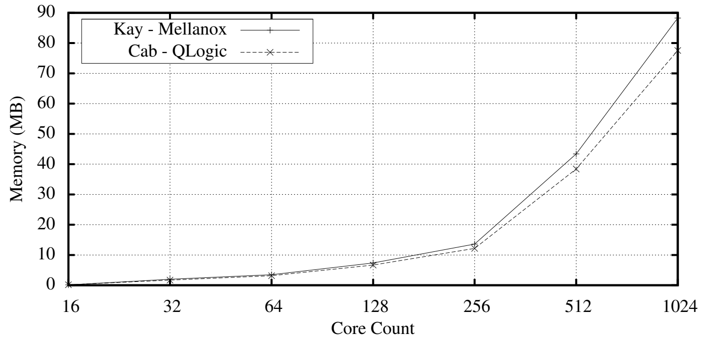
Listing 6.1 illustrates the output of the HWM functional breakdown, displaying a partial call stack for clarity, for the HWM thread on a 1024 core run of Orthrus with OpenMPI on Kay.

From this is is clear to see that MPI is consuming 37% of the total memory at the point of HWM. We also see an example of this consumption as the top consuming call stack, accounting for ≈27% of memory, occurs from the PMTM library calling an MPI function. As the highest consuming function is MPI related, we can immediately tell that MPI consumption is problematic in this application configuration.

By plotting the temporal memory consumption of 'libmpi.so' for the same



(a) Memory HWM



(b) MPI memory at HWM

Figure 6.5: Platform comparison of Orthrus memory scalability with OpenMPI

1024 core run, we can ascertain the nature of the consumption. Figure 6.4 shows us that the MPI memory consumption is near constant (at ≈75 MB) throughout the duration of the run. We note that this figure does not include memory allocated by other MPI libraries, such as the 'openmpi/mca_*.so' functions, which account for an additional ≈15 MB.

This indicates that there is something inherently inefficient about the way OpenMPI is handling memory throughout the execution, rather than a particular spike in consumption due to a specific operation. Whist this is of serious concern it does indicate a real opportunity to reduce memory consumption, if a more memory efficient MPI configuration can be utilised.

6.5 MPI Implementation Comparison

Figure 6.5 provides a comparison of the two platforms for the memory consumption of our benchmark application, as it is weak scaled. It allows us to compare the memory consumption of both platforms without the use of any optimisation libraries. Whilst there is significant growth in memory as we weak scale, we can not fully attribute this to poor MPI scalability, as changes in the global problem during scaling may result in increased consumption. To understand the actual MPI memory scaling behaviour we must employ WMTools to decompose the sources of consumption at the point of HWM.

Figure 6.5(b) illustrates the scaling of the memory attributed to MPI at the point of HWM. What we can observe from this graph is that the scaling of MPI memory is very poor in both cases, consuming up to 88 MB at 1024 cores. There is a noticeable difference between the MPI memory consumption of QLogic and Mellanox, of up to 14%.

The most concerning artefact of this graph is not the memory consumption numbers, but the scaling. When we scale the core count from 16 to 1024 cores (a 64× increase in processes) we see a corresponding increase in MPI memory consumption of 616× with Mellanox on Cab. This behaviour is also represented on the QLogic platform where we observe a 470× memory increase for the same increase in processor count. This growth rate is unsustainable, even for current job sizes, due to the limitations of memory capacity. The challenge is then scaling Orthrus beyond tens of thousands of cores, without MPI dominating memory consumption.

6.6 Runtime Configurations

In this section we look at a platform specific MPI implementation, BullXMPI, a derivation of OpenMPI tuned specifically for Mellanox InfiniBand. Whilst BullXMPI already contains optimisations for both performance and memory consumption, it is also supplied with a number of runtime profiles. These runtime profiles each tune a specific aspect of behaviour, such as: packet size, quality of service and RDMA configurations. A low memory footprint profile is available to further address memory consumption, and can be used to tune the size, quantity and type of receive queues used in communication.

The default platform configuration utilises both per-peer (P) buffers and shared buffers (S). Receive queues are specified as a list of different queue types, in order of increasing size. The format is as follows:

$$\begin{gathered} < QueueType >, < BufferSize >, < BufferCount >, \\ < MinimumBufferCount >, < SendAllowance > \end{gathered} \tag{6.1}$$

The default receive queue configuration for OpenMPI on InfiniBand is:

$$\begin{gathered} P, 128, 256, 192, 128 : S, 2048, 256, 128, 32 : \\ S, 12288, 256, 128, 32 : S, 65536, 256, 128, 32 \end{gathered} \tag{6.2}$$

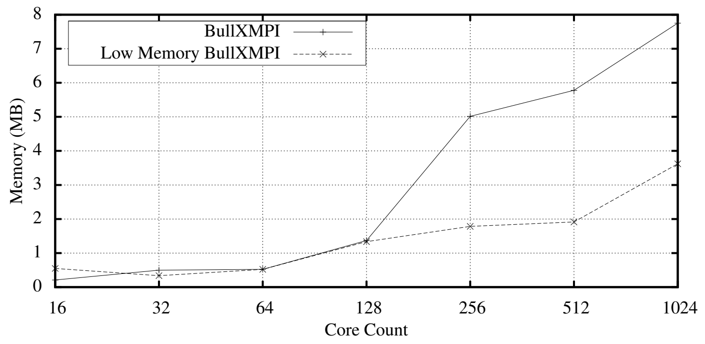
Our optimised BullXMPI configuration utilises the Mellanox XRC (X) buffer type, with a very simple layout:

$$X, 6144, 128, 64, 1 \tag{6.3}$$

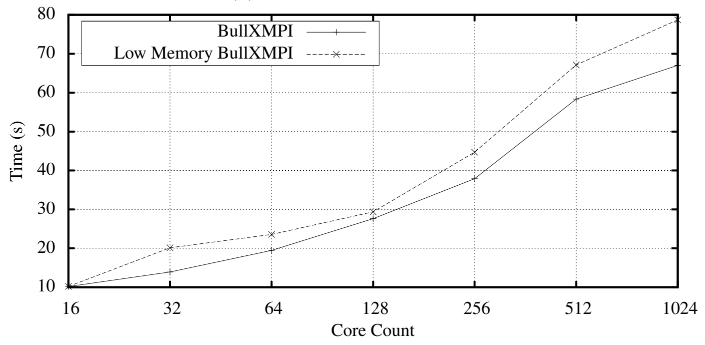
This receive queue configuration is coupled with a `btl_openib_max_send_size` value of 6144.

If we relate the default OpenMPI queue configuration (Equation 6.2) back to our message size histogram (Figure 6.2) we can see that all of the small messages will be captured by the P buffer, and a number of the larger messages by the S buffers. Although the largest messages will have to be split to fit in even the largest S buffers, there is generally good coverage for the message sizes of Orthrus.

The new XRC queue structure for the low memory profile (Equation 6.3) does not provide good coverage of the message sizes. At ≈6 KB the queue is wasteful for the smallest of messages, and insufficient for the larger messages which are naturally split as a result of the reduced maximum send size. Whilst



(a) MPI memory at HWM



(b) Runtime

Figure 6.6: Orthrus BullXMPI comparison with low memory profile on Kay - Mellanox

this new structure should provide very strict memory behaviour we expect it to have a negative impact on the performance of Orthrus, due to the inevitable queueing of messages resulting from insufficient buffer space.

Whilst our analysis leads us to believe that the configuration provided by the Bull profile is non-optimal for Orthrus in this configuration, we could use our understanding to design an optimal configuration. Any such configuration would be specific to the application run in that configuration of problem size and core count, and would not be generally applicable. Thus analysis would be required to generate the best queue configuration for that specific execution, which is not a viable method of optimisation.

Figure 6.6 shows the impact of running a platform specific MPI implementation, BullXMPI. The use of BullXMPI presents a significant reduction in MPI memory consumption over OpenMPI (11.4× reduction at 1024 cores), as can be seen by comparing Figure 6.6(a) and Figure 6.5(b).

We can also see that the use of XRC receive queues, in the low memory profile, affords an additional MPI memory reduction over BullXMPI (2.1× reduction at 1024 cores). This represents a 24× MPI memory reduction over OpenMPI.

From Figure 6.6(b) we can see that there is a performance impact for utilising the low memory profile in BullXMPI (1.18× slowdown at 1024 cores). We observe that this slowdown is more profound at larger scale, observing a similar trend in the memory savings. We attribute this performance loss to the queueing of messages when insufficient receive buffer resources are available.

6.7 Vendor Libraries

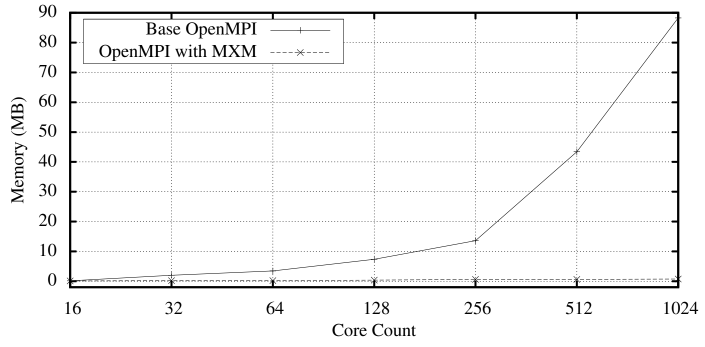
In this section we look at the effects of vendor-optimised libraries for the MPI stack. We study how these libraries, when utilised on the relevant platform, improve performance of OpenMPI with regards to both memory consumption and runtime.

Whilst QLogic have provided an optimisation library for OpenMPI for a number of years, the recent introduction of the Mellanox equivalent makes this a pertinent investigation.

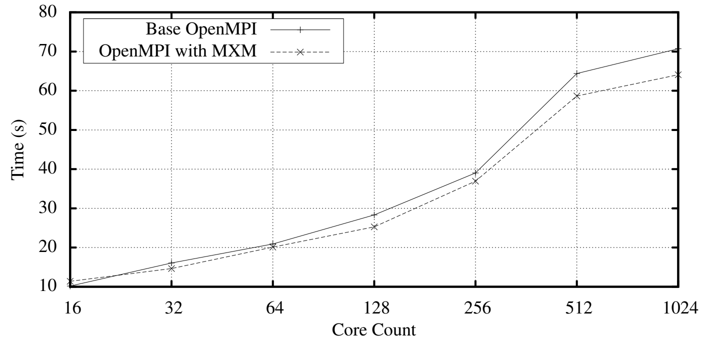
6.7.1 MXM

Figure 6.7 illustrates the scaling of the MPI memory consumption and application runtime, of the base OpenMPI install and the MXM configuration. It is clear to see from Figure 6.7(a) that at 1024 cores the MXM library offers a significant reduction in MPI memory over the standard MPI install; from 88 MB to 0.75 MB (a 117× reduction).

The key aspect of these results is the near-flat scaling of MPI memory consumption from the MXM configuration. We witness only a 7.4× increase



(a) MPI memory at HWM



(b) Runtime

Figure 6.7: Orthrus using MXM comparison with default OpenMPI on Kay - Mellanox

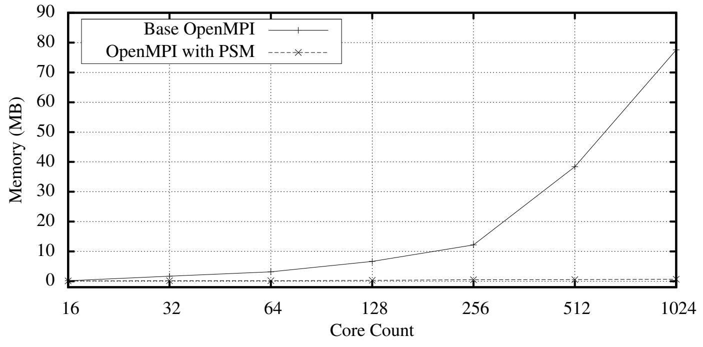
in consumption through the 64× increase in core count, as opposed to the 616× increase exhibited by the standard OpenMPI configuration.

From Figure 6.7(b) we observe a very similar performance trend between the standard MPI install and the MXM optimised version; on average the MXM configuration is ≈1.05× faster.

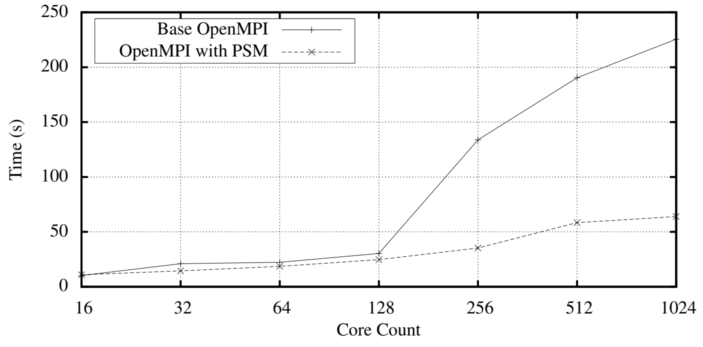
6.7.2 PSM

Figure 6.8 demonstrates the scaling of the MPI memory consumption and runtime, of the base OpenMPI install and the PSM configuration. Similar to Figure 6.7 we see a significant improvement of MPI memory consumption at scale – 115× reduction at 1024 cores.

The performance difference between the default configuration and the PSM



(a) MPI memory at HWM



(b) Runtime

Figure 6.8: Orthrus using PSM comparison with default OpenMPI on Cab - QLogic

configuration, shown in Figure 6.8(b), is substantial. This is to be expected as the PSM library is also designed to address the processing of messages, unlike MXM which is specifically designed for message management. On average PSM provides a performance improvement of ≈2×, a significant advantage.

6.8 Application Modifications

In the process of performing the application analysis we identified an unusual communication pattern within the Orthrus benchmark application. Whilst the underlying algorithm should make use of basic seven-point stencil communication, where each process communicates with its six surrounding processes in the 3D cube, we did not see this reflected in the communication analysis (shown in Figure 6.1). This discrepancy suggested that PETSc was not utilising the data structure of the matrix in the intended way.

Subsequent modifications to Orthrus, by researchers at AWE and Warwick, to address the storage of ghost cells and to utilise the PETSc structured interface allow the data layout to be fully conveyed. Since these modifications we now see a standard seven-point stencil communication pattern, and a reduction in point-to-point communication messages sizes. As a result of these changes we observe a ≈15× reduction in application memory consumption at HWM.

6.9 Summary

In this chapter we have shown how WMTools can be used to drive an investigation into memory consumption scalability. Using trace data, from WMTrace, we have been able to decompose memory at the point of HWM, attributing allocations to their source function or library. From this we have been able to track the memory consumption of MPI as we weak scale a benchmark application.

We used this data to illustrate the poor scalability of MPI memory consumption on InfiniBand platforms with OpenMPI, demonstrating a 616× increase in consumption when scaling from 16 to 1024 cores (on Mellanox hardware).

We were able to track changes in memory consumption when we employed a vendor-specific MPI implementation, BullXMPI. By relating characteristics in the application's communication and memory profiles to the layout of communication buffers we were able to pre-empt behaviour when transitioning to a new queue structure. Using this analysis we believe it would be possible to engineer application-specific receive queue structures to optimise memory consumption, whilst minimising impact on application performance.

Further, we were able to show how vendor-specific libraries can be used to configure OpenMPI implementation to vastly improve MPI memory consumption (showing a 117× reduction on 1024 cores on Mellanox hardware). The analysis capabilities of WMTools facilitate the validation of such memory reduction techniques, by providing a clear breakdown of the source of consumption. Our analysis showed how memory consumption can be reduced by the interchanging of MPI libraries, without modification to the underlying application.

Where vendor-specific optimisation libraries are not currently available this process of analysis – of first identifying a problem external to the application and then validating the success of the potential solution – can be used to justify the provisioning of such libraries to users.

CHAPTER 7
Memory Modelling

WMTrace is a lossless data collection tool which stores all data in a file. One of the novel uses for this information is comparative analysis between processes of different runs.

In Chapter 5 we demonstrated how memory scaling is not always linear with core count. This idea was developed in Chapter 6 where we investigated the memory consumption of MPI libraries showing how memory can increase with scale. Combined with the analysis presented in Chapter 5 we can see that memory scaling is more complex than immediately obvious. Strong scaling will reduce the local domain's memory consumption, whilst weak scaling will keep it constant. At the same time, there are other factors at play which will influence the scalability such as ghost cells, constant data and MPI consumption. Combined these factors make it harder to predict memory consumption at scale.

In this chapter we discuss the construction of memory consumption models, through the analysis of multiple trace files. We deduce that at any point in time we know the composition of memory, and the location of each allocation. By comparing two or more traces we can identify corresponding allocations, and look for behavioural patterns in allocation size.

Using some basic information about the different executions, specifically job size and problem size, we can relate size changes in corresponding allocations to the changes in job configuration. Whist not 100% accurate, we can identify some significant trends of growth, and use these trends to generate a model for memory consumption.

These models can be used to speculate on memory consumption for jobs of different core counts or input size. This information can be incredibly valuable in understanding if a job will fit into memory. Analysis of the model can also help identify problems in application memory consumption scalability.

We show how, by using the models, we can experiment with different runtime configurations than those currently available in the application. From these we speculate on the available memory savings afforded by both 3D processor decompositions and hybrid parallelisation models in the Chimaera benchmark application.

In this chapter we provide memory models for Lare2D and Chimaera, these applications have been chosen for this analysis due to their characteristics. The Lare2D benchmark represents a very simplistic 2D grid calculation, and thus is easy to understand, and model. As the primary memory consumption occurs from variables on the grid it has predictable scalability characteristics, and thus is a good candidate to demonstrate the modelling capabilities within WMTools.

Chimaera represents a more complex 3D grid problem. It has also been the candidate of our previous memory consumption analysis techniques earlier in this thesis. This allows us to use our developed knowledge of the application structure, and memory behaviour, to develop a more complex memory scalability model.

Although we only demonstrate the modelling capabilities on these two applications, the methods are applicable to a much wider range of applications. Whilst we make no guarantee of the accuracy of this modelling technique outside of the evaluated scope, we feel confident that it can be successfully applied to alternative applications.

The initial concepts of memory modelling presented in this chapter were first published in [103].

7.1 Related Work

There is a history of parameterised performance models at Warwick [29, 91]. Such models are designed to utilise application instrumentation to understand the performance weightings of certain application operations. Knowing the application input parameters and the decomposition then allows you to predict application runtime for arbitrary scale with a high degree of accuracy.

In addition to parameterised models we have utilised simulation based models to predict application runtime [46]. Here an application skeleton is used with a few specific parameters, such as processor and network performance metrics, to emulate execution.

Both these approaches to modelling are very labour intensive, and require deep knowledge of the application. The resulting models are also specific to application and often input deck. Our approach is different, because it uses automated comparison of data points. Without the use of domain knowledge there is naturally a lower level of accuracy and flexibility in these models, but they can be generated much quicker and with increased ease.

There has been a body of research into predicting heap memory consumption, under different circumstances. In [51] Hofmann and Jost present a method of predicting heap usage for first order functional languages. Using linear programming, and type deviation, they determine the use of function parameters, and use this information to infer heap consumption.

Similarly Braberman *et al.* develop a parameter based heap estimation methodology for Java based applications [16]. Their approach uses call chains to track parameter propagation, for a set of values within a method of specific interest.

Both of these approaches are designed to estimate an upper bound of dynamic memory usage, based on the values of input parameters. Our approach avoids static source code analysis but rather looks at trends which actually occur when the application is run in different configurations. The two parameters of particular interest then become local problem size and core count, rather than specific function parameters. We understand that our method is likely to be less accurate than a fully parameterised model, as it is constructed from a specific instance – generating an approximate scaling function.

7.2 Point-wise Comparison

The first step towards a memory model is establishing how each allocation at the HWM, changes as the problem is scaled. For this analysis we must take two traces from runs at different scale and perform a point-wise comparison between allocations. We must look at how each allocation size has changed, in relation to the change in problem size and core count.

Firstly, WMAnalysis looks at the call stack of each allocation at the point of HWM and maps them to the corresponding call stacks of the other trace. This mapping then allows us to visualise the comparable memory consumption for the two traces. For each call stack we can then compare the allocation values in the two traces, to build up a mapping of proportional change in allocation size.

We then search for certain specific relationships by looking at these allocation ratios in comparison to ratios of local problem size and core count. At a basic level we identify pairs of allocations where: the allocation size is constant, the change is proportional to the change in problem size, and the change is proportional to the change in core count. WMAnalysis can then aggregate this information and produce a general formula to express the changes in memory consumption as the problem is scaled.

$$F(P, N) = C_1 \frac{P}{N} + C_2 N + C_3 \tag{7.1}$$

Equation 7.1 represents the three components of the memory model for problem size P and core count N. The first constant, C_1, represents the portion of memory that shrinks when the core count is increased, this generally represents the local problem domain. C_2 represents the portion that increases with the core count, such as communication buffers. Finally, C_3 represents the volume of memory which remains constant, regardless of changes in problem size or core count, representing the static memory of the problem.

7.2.1 Linear Regression

To establish these memory site allocations we employ a very simplistic process of linear regression. Using the ratio of two allocation sizes (their gradient) we compare with a set of expected ratios. Such an analysis between two data points on their own is fairly unreliable, but extremely quick and simple. Our methodology gains accuracy by the volume of data accumulated. At the point of HWM there may be tens or hundreds of thousands of live allocations, each representing a pair of points to evaluate.

So whilst the accuracy of a single linear regression may be low, the accumulated result of thousands of these relationships becomes a slightly more accurate prediction of behaviour.

The analysis of more than two points, from three or more trace files could help improve the accuracy of the linear regression, but it is more complex to perform this evaluation.

7.2.2 Misinterpretation of Relationships

The process of grouping allocations makes many assumptions about the source of the allocations and their variable dependencies. As a result there can be errors in categorisation, primarily due to a lack of data.

With only two data points for the analysis of each allocation, WMAnalysis essentially performs a linear regression and compares the result with known gradients. Many allocations exhibit more complex behaviour, thus we must make some allowances when matching allocations.

One of the biggest pitfalls with this approach of modelling is the expectation that relationships which exist in the example case will persist in the general case. Allocations which halve in size between two analysed runs are therefore assumed to halve in a repeating pattern for further runs. This makes it extremely hard to accurately model data decompositions which change shape on different core counts.

	High Water Mark (MB)
2	2259.91
4	1138.13

Table 7.1: High water mark results for `Lare2D` 4096^2

WMAnalysis attempts to addresses this by interpreting processor decomposition information, but is still prone to incorrect inferences under certain conditions. Typically a doubling of the core count will halve the local problem domain in the longest dimension, to achieve the most regular decomposition. This means that dimensions exhibit a stepping function of reductions, in accordance with the processor decomposition. Thus when taking two consecutive strong scaled results the buffer size of only one dimension will have changed, from this the model can only conclude that this is the only dimension with a buffer size proportional to local problem size. Predicting this stepping function is unreliable from only two traces, and would require a more complex model built from three or more trace files. Such functionality is still under development within WMTools.

We also note that this methodology is only applicable to deterministic executions. Where there is no deterministic decomposition, such as AMR-based applications (which employ a load-balancing algorithm) or in Partice-In-Cell (PIC) codes [124], such a modelling technique is not necessarily appropriate.

7.3 Lare2D - Simplistic Model

In this section we show the progression of the automated modelling technique. We start by generating a model for a simplistic 2D magnetohydrodynamics (MHD) application, Lare2D, with good memory scaling. This enables us to demonstrate the methodology working in a controlled environment.

Due to the simplicity of Lare2D we attempt to model the strong scaling of a large problem size 4096^2. We do this by analysing the HWM traces from execution on two and four core runs; the HWM values for these runs are shown

	Prediction (MB)	Actual (MB)	Error (%)
1	4505.22	4495.02	0.22
2*	2259.88	2259.91	-0.00
4*	1137.22	1138.13	-0.08
8	575.89	577.37	-0.26
16	295.23	296.47	-0.42
32	154.91	149.36	3.70
64	84.78	85.67	-1.04
128	49.75	58.98	-15.64
256	32.34	42.32	-23.59

Table 7.2: Model prediction results for `Lare2D` 4096^2

in Table 7.1.

We can see that at this scale the memory scaling is highly efficient (1.99× reduction in HWM for a 2× increase in problem size). Whilst this does not express any of the more complex behaviour the code may exhibit at scale, it is suggestive that our model will be largely based around a local problem size value.

$$F(P, N) = 280.7\,\frac{P}{N} + 1016\,N + 15257783 \tag{7.2}$$

The model generated from these two trace files is shown in Equation 7.2. As we can see Lare2D has very good memory scalability; only a very small component increases with core count, and the constant consumption is relatively low (14.5 MB). As predicted, there is a large component dedicated to local problem size, indicating that the code should scale well.

Using the model we can then predict the memory consumption of Lare2D at scale. Table 7.2 shows these predictions validated against experimental results. We see that the model error is very small, with a slight under-prediction, until 128 cores, where the error jumps to over 15% indicating a change in behaviour.

We can see from Figure 7.1 that our model prediction for Lare2D 4096^2 at 16 cores is accurate for the HWM. We can also see that there are three distinct phases to the execution: an initial startup phase, a compute phase and finally a concluding phase. The start and end phases represent the problem composition

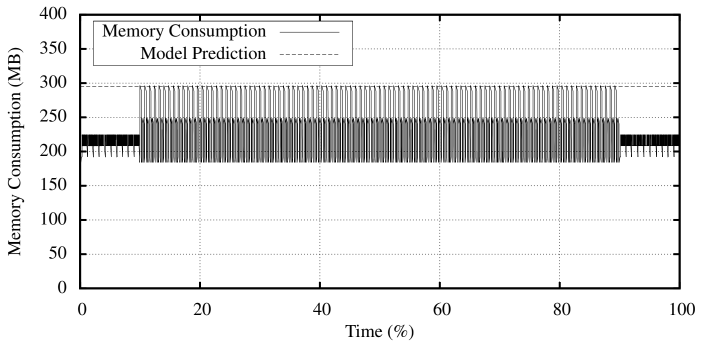


Figure 7.1: Model prediction against temporal trace of Lare2D on 16 cores

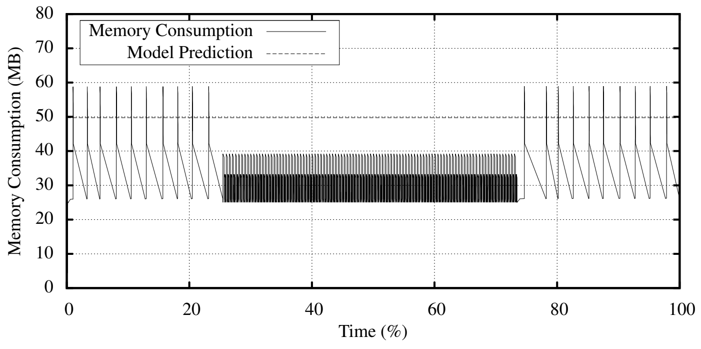


Figure 7.2: Model prediction against temporal trace of Lare2D on 128 cores

and I/O operations of the application, rather than the actual compute phase.

If we plot one of the higher core count runs, where our model predictions were less accurate, such as 128 cores (Figure 7.2) we get an insight into the cause of model inaccuracy. The previously memory dominant phase, the compute, is now shadowed by the surrounding I/O phases. Whilst still inaccurate at this scale, the model is actually predicting the memory consumption of the compute phase, rather than this newly dominant I/O phase. Our model is, in fact, over predicting the compute phase by ≈10 MB, suggesting that we are not factoring in sufficient scalability.

	Prediction (MB)	Actual (MB)	Error (%)
64*	85.55	85.67	-0.14
128*	54.71	58.98	-7.23
256	39.35	42.32	-7.02

Table 7.3: Second model prediction results for `Lare2D` 4096^2

7.3.1 Multiple Models

By modelling the I/O phase and the compute phase separately, by invoking the analysis tool on different runs, we can generate a second model for Lare2D. We can then simply take the maximum of these models, for any given scaling point, as our prediction.

$$F(P,N) = 246.9\,\frac{P}{N} + 566.5N + 24939843 \tag{7.3}$$

Equation 7.3 represents our new model for the I/O phase, based on the 64 and 128 cores runs. Their validation is presented in Table 7.3, and presents a significant improvement over those in Table 7.2.

We note that we first observed this I/O phase at very low core count, but did not anticipate it to exhibit a different scaling behaviour to the compute phase. Another approach would have been to generate three models, from our initial traces, one for each of the obvious phases. This would allow us to plot the behaviour of the of the phases over time, however our approach of sampling the traces when the new phase becomes dominant allows us to be more confident about the magnitude of this consumption.

A future extension of these capabilities would be to allow the user to specify regions of interest and generate models solely for those regions.

7.3.2 Increased Problem Size

Using our new compound model we can now attempt to predict the memory consumption of a larger problem (8192^2). Table 7.4 validates our predictions for this new problem size, and we observe a generally high level of accuracy. As

	Prediction (MB)	Actual (MB)	Error (%)
1	17977.43	17942.23	0.20
2	8995.99	8978.81	0.19
4	4505.27	4495.13	0.23
8	2259.91	2253.38	0.29
16	1137.24	1131.62	0.50
32	575.92	570.83	0.90
64	295.28	289.97	1.83
128	155.01	149.64	3.59
256	85.65	86.18	-0.61

Table 7.4: Compound model prediction results for `Lare2D` 8192^2

we have increased the global problem size, it is easier for the models to track the memory consumption, as a higher percentage of memory is consumed by the local problem.

Our simplistic model of Lare2D does not capture the behaviour of ghost cells, but rather encompasses them within the local domain behaviour. This failure to capture the nuances of behaviour are more problematic in the modelling of smaller problems, as the ghost cells make up a larger percentage of memory consumption. As we move to the larger problem, there is a reduction in the ratio of ghost cells to data cells, thus increasing the accuracy of our model. With a 4096^2 problem on 256 cores, we have a local problem size of 256×256 cells with a local boundary of 1024 cells; for the 8192^2 problem, we have a local problem size of 256×512 cells with 1536 boundary cells. Thus the ratio of volume to boundary cells has increased from 192:3 to 256:3, giving more efficient memory utilisation and making it easier for our basic model to predict memory consumption, without the consideration of ghost cells.

7.4 Chimaera - Complex Model

As we have seen in Chapter 4, Chimaera has reasonably complex memory scaling behaviour with strong influences from MPI and decomposition. For this reason we develop a more comprehensive model to predict behaviour in more depth.

To ensure that we capture the MPI behaviour we train our model on the 32 and 64 executions of the 120^3 problem, as these core counts are big enough to exhibit some MPI artefacts.

$$F(P,N) = 14023\,\frac{P}{N} + 105887\,N + 1584851 + 11081.9(GhostCells) \qquad (7.4)$$

From these traces we generate the model shown in Equation 7.4. We note that the extra term used in this equation is used to denote the memory attributed to ghost cells. WMAnalysis assumes a single halo of ghost data and identifies the scaling relationships proportional to the changes in ghost cell size to automatically generate this term. Enhancing WMAnalysis to account for multiple levels of ghost data would be trivial, but would require the user to specify the code behaviour at the point of model generation. We also note the use of the overly simplified term $\frac{P}{N}$, which assumes that the global problem is evenly distributed on every processor. Whilst this is true for very large problem sizes, and low core counts, we have already shown it not to be the case in Chimaera. For this reason we manually calculate the actual processor decomposition and pass it to the model, with the corresponding local domain size. Whilst this is a more involved process, it will produce a more accurate model prediction.

Table 7.5 presents the validation of the model, from Equation 7.4, against measured HWM values. From Table 7.5(a) we can see that the model is generally very accurate for the 120^3 problem, with only a minor lapse at 512 cores which amounts to a difference of 6.9 MB.

Using the same model we can predict the memory consumption of different problems, as the underlying behaviour of the application should be consistent. Table 7.5(b) validates our model against the results of the 60^3 problem. We can see from the error that there is a factor which our model is failing to capture, though the magnitude is still reasonably small, 8.1 MB for 256 cores.

	Decomposition	Local Cells	Ghost Cells	Prediction (MB)	Error (%)
16	4x4	108000	16928	1626.36	n/a
32*	4x8	54000	12368	857.62	-0.31
64*	8x8	27000	8258	456.33	-0.16
128	8x16	14400	6340	274.02	-0.33
256	16x16	7680	4520	177.84	-0.95
512	16x32	3840	3480	141.35	-4.64

(a) Chimaera 120^3

	Decomposition	Local Cells	Ghost Cells	Prediction (MB)	Error (%)
16	4x4	13500	4418	230.36	-0.03
32	4x8	7200	3340	136.33	-0.81
64	8x8	3840	2360	84.27	-4.99
128	8x16	1920	1800	59.14	-10.02
256	16x16	960	1272	53.64	-13.14
512	16x32	480	1008	70.29	n/a

(b) Chimaera 60^3

Table 7.5: Model validation for Chimaera

From both problem sizes in Table 7.5 we can analyse the ratio of local data cells to ghost cells, at large scale. For the 60^3 problem on 512 cores we see over twice as many ghost cells than data cells, and an almost 1:1 ratio for the similar case on the 120^3 problem. This emphasises the importance of capturing ghost cells as a model parameter, though knowledge of the domain decomposition algorithm is required.

7.4.1 Growth Factor Analysis

One of the most important elements of the model is the component relating to core count, as this will determine the increasing factor as the core count is scaled (represented as the C_2 term in Equation 7.1).

The majority of this consumption is likely to originate from the MPI library unless the application manages its own rank-to-rank communication buffers. This means that we can visualise the model's prediction against that of our two problem sizes, to gain an insight into the accuracy of our trend prediction. Whilst the model term is not designed to specifically model MPI growth, and will not factor in constant size allocations, it is a good approximation.

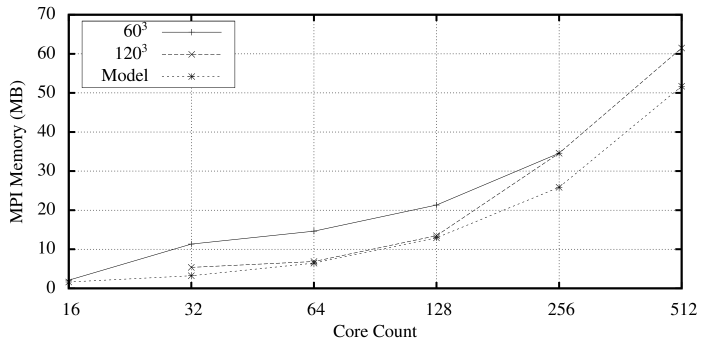


Figure 7.3: Chimaera MPI memory growth against model prediction

Figure 7.3 shows that there are essentially two magnitudes to the experienced MPI memory consumption. MPI memory consumption should be dependent on core count, and roughly problem size independent, thus we would expect both problem sizes to exhibit roughly the same MPI memory consumption.

What we see from Figure 7.3 is that our model predicts, with reasonable accuracy, the trend of growth but arguably fails to grasp the magnitude. One potential reason is that we trained our model on the 120^3 problem on 32 and 64 cores, which from the graph do not exhibit the same magnitude as the equivalent sizes for the 60^3 problem.

To experiment, we retrain our model on the HWM traces from the 32 and 64 core runs for the 60^3 problem and revalidate. An alternative would have been to utilise the higher core count runs (256 and 512) from the 120^3 problem which also exhibits the increased consumption. Our approach, of using the 60^3 problem, shows how accurate models can be generated from varying problem sizes.

$$F(P, N) = 12480.2\,\frac{P}{N} + 108125\,N + 9894469 + 10968.9(GhostCells) \tag{7.5}$$

Equation 7.5 represents our updated model, based on the 60^3 problem traces. We can see that this model has a vastly increased static term, which will increase

	60^3		120^3	
	Prediction (MB)	Error (%)	Prediction (MB)	Error (%)
16	217.98	-5.40	1473.59	n/a
32	133.37	-2.97	784.82	-8.77
64	86.43	-2.56	423.78	-7.28
128	64.32	-2.14	260.35	-5.30
256	60.57	-1.93	174.52	-2.79
512	78.49	n/a	144.34	-2.63

Table 7.6: Model predictions for Chimaera using Equation 7.5

predictions by ≈8 MB, accompanied by a reduction in local problem size.

Table 7.6 shows the predictions, and their associated error, for both problem sizes when modelling is based on Equation 7.5.

In comparison to the results in Tables 7.5(a) and 7.5(b) the general error rate is a a bit higher, but in contrast to the previous model our accuracy actually increases at scale. This trend is likely to be the result of the reduction in the local problem component, which plays a more important role for larger problems and at small scale. This means that for a few allocations the model is incorrectly identifying a relationship as constant, where it is actually proportional to problem size.

7.5 Modelling Implementation Changes

In this section we make two conjectures about the design of new features within the Chimaera code, and use our models to investigate their properties. From our study in Section 5.4 we established the importance of processor decompositions and hybrid parallelism models in reducing ghost cells and improving memory consumption scalability.

Here we will apply modifications to our models to simulate the implementation of these features and make conjectures about the resulting memory consumptions. As such we are unable to validate these results, and play no consideration to implementation design choices, but rather model based on theoretical savings. Additionally we make no comment on the performance of any such

	Decomposition	Local Cells	Ghost Cells	Prediction (MB)	Predicted Saving (%)
16	4x2x2	13500	3908	212.64	2.45
32	4x4x2	6750	2498	119.21	10.62
64	4x4x4	3375	1538	72.29	16.35
128	8x4x4	1800	1090	55.46	13.77
256	8x8x4	960	740	55.00	9.19
512	8x8x8	512	488	73.43	6.45

(a) Chimaera 60^3

	Decomposition	Local Cells	Ghost Cells	Prediction (MB)	Predicted Saving (%)
16	4x2x2	108000	15008	1453.50	1.36
32	4x4x2	54000	9488	754.70	3.84
64	4x4x4	27000	5768	397.73	6.15
128	8x4x4	13500	3908	224.19	13.89
256	8x8x4	6750	2498	142.30	18.46
512	8x8x8	3375	1538	118.49	17.91

(b) Chimaera 120^3

Table 7.7: Model predictions for Chimaera with 3D processor decomposition

implementations.

We base our further analysis on the architecture of the Cab platform, in accordance with the Chimaera model generated in Section 7.4.

7.5.1 3D Processor Decomposition

A 2D decomposition of a 3D problem domain will result in local problems in the shape of a cuboid (a 3D rectangle). Utilising a 3D processor decomposition will enable the generation of more regular cubic shapes. As we demonstrated in Section 5.4 the closer to a regular cube the lower the surface to volume ratio, thus minimising ghost cells.

For both the 120^3 and the 60^3 problem we simulate the best 3D processor decompositions and use these to generate memory predictions based on the model in Equation 7.5, and generate an estimated memory saving from the model results presented in Table 7.6; these predictions are presented in Table 7.7.

If we study the balance of ghost cells to problem cells with our 3D processor decomposition, against the standard 2D decomposition, we can see a vast improvement. For the 512 core, 120^3, case we observe 3840 local cell and 3480 ghost cell for the 2D decomposition (Table 7.5(a)) and 3375 local cells with 1538 ghost cells for the 3D decomposition (Table 7.7(b)), a $\approx 2\times$ increase in the ratio of local cells to ghost cells. This trend of improvement is exhibited across the experiments when comparing the 2D and 3D decompositions, though is more pronounced at higher core counts where difference between the 2D pencil and the newly established cubic shape is most extreme.

We note that in certain circumstances, such as the 512 core 60^3 problem (Table 7.7(a)), the 3D decomposition results in more local cells than the comparative 2D decomposition. This is a result of the decomposition of non-power-of-two problem sizes onto power-of-two processor counts. Fortunately we also see an approximate halving of ghost cells, thus an overall memory reduction is still achieved.

As a whole the memory savings presented in Table 7.7 are significant, and if they were implemented in a sufficiently performant configuration, could prove very beneficial.

Increased Scale

Using a 3D processor decomposition has one additional benefit: the ability to scale to more processes. A 2D decomposition of 60^3 cannot scale beyond 3600 cores, as this would represent a $1 \times 1 \times 60$ problem decomposition; more cores could not decrease the local problem size, and would be wasted. Using a 3D decomposition, it would be theoretically possible to scale to the maximum 216000 cores where a $1 \times 1 \times 1$ problem decomposition would be achieved.

7.5.2 Hybrid MPI and OpenMP

For the hybrid modelling we remain with the 2D decomposition, and use the layout developed in Section 5.4 of 1 MPI process and 8 OpenMP tasks per socket. As such our modelling of 16 cores will now represent the execution of 2 processes, with the underlying parallelism of 16 OpenMP tasks.

As we crudely model the memory consumption, without any consideration of

	Decomposition	Local Cells	Ghost Cells	Prediction / Socket (MB)	Prediction / Core (MB)	Predicted Saving (%)
16	2x1	108000	15008	1453.50	181.69	14.56
32	2x2	54000	9488	754.70	94.34	20.86
64	4x2	27000	6728	407.77	50.97	29.49
128	4x4	13500	4418	229.53	28.69	48.27
256	8x4	7200	3340	156.47	19.56	64.44
512	8x8	3840	2360	132.62	16.58	77.42

(a) Chimaera 60^3

	Decomposition	Local Cells	Ghost Cells	Prediction / Socket (MB)	Prediction / Core (MB)	Predicted Saving (%)
16	2x1	864000	58808	10909.63	1363.70	6.18
32	2x2	432000	36968	5541.13	692.64	8.22
64	4x2	216000	26048	2859.36	357.42	10.13
128	4x4	108000	16928	1485.14	185.64	17.20
256	8x4	54000	12368	807.92	100.99	29.03
512	8x8	27000	8258	469.97	58.75	50.42

(b) Chimaera 120^3

Table 7.8: Model predictions for Chimaera with hybrid parallelism

implementation-specific tuning, we consider our predictions to be a lower bound of consumption.

As our new memory predictions will be per-socket, we must compare them with the equivalent figures from the model on a per-core basis, thus we divide our answer through by eight, to represent the number of cores-per-socket. Whilst a per-node comparison would be fairer, since in the normal decomposition not all cores have the same memory consumption, it would depend on rank placement to determine socket and node level memory consumption.

As with our 3D decomposition model, our predictions are based on our existing memory model (Equation 7.5), and our predictions make no comment on the computational efficiency of a hybrid solution.

Table 7.8 presents our predictions for memory savings through the use of hybrid parallelism. We predict the memory-per-socket and derive a memory-per-core figure by even decomposition, and compare with the predictions from Table 7.6.

A number of different memory reductions are achieved by this technique, which add up to a very significant memory saving in certain cases. Firstly, the decompositions are performed on socket counts, thus lower numbers by a factor of eight, enabling more balanced decompositions than the equivalent core count based decompositions. Secondly, by having a lower decomposition count we remove all internal ghost cells on the socket, which can be very significant at large scale. Lastly we avoid the duplication of constant data, across each core within the socket, represented by the C_3 term in Equation 7.1.

As we can see from Table 7.8(a) this can theoretically amount to a memory reduction of over 75%, for the 60^3 problem on 512 cores. Whilst we would not expect to see quite this magnitude of memory savings in a production implementation, they are not that distant from those presented in Section 5.4. In that example we measured the effects of hybrid parallelism on SNAP (Figure 5.7(a)) and demonstrated a 43% reduction in memory consumption at 1024 cores for a real implementation of this technique.

Amalgamation of Techniques

As discussed in Section 2.1, core count densities are expected to keep rising. As they do so, hybrid parallelisation models will become increasingly appealing, if only for the memory reductions demonstrated above.

Whilst not presented, there is no reason that the two techniques demonstrated in this chapter (3D decompositions and hybrid parallelism) could not be combined for additional memory savings. Whilst the performance characteristics of such an implementation are currently unknown, the potential memory reductions presented should warrant further investigation into application development. As memory restrictions play a more critical role in application design, a tradeoff between runtime and memory is likely to be required.

7.6 Model Confidence

In this chapter we have shown how, with WMTools, we can automatically generate memory consumption models. Our technique requires the two sizes of execution to train the model, after which we are able to predict for different problem sizes and core counts.

To demonstrate this we have used two different benchmark applications, and in both situations been able to validate the model predictions at larger core counts, and problem sizes, with a high degree of accuracy. Although such results do not establish a universally high degree of model confidence they demonstrate the success of the methodology. Modelling, and validating the associated results, for a significantly wider range of applications would increase confidence in the methodology, but was beyond the scope of this research.

For both our applications we were required to validate the model predictions on a few more test cases before we were confident of the large scale predictions. Such a validation methodology would be advised for any subsequent application modelling exercises.

7.7 Summary

In this chapter we have presented the methodology of generating memory models from application traces. Using WMAnalysis we have demonstrated our implementation on two different applications (Lare2D and Chimaera). To the best of our knowledge this is the first demonstration of the automatic generation of a memory model based on application traces.

We have shown that the data collection and storage methodology employed by WMTrace facilitates a much deeper level of analysis than is available in any other comparable tool, thus justifying the runtime overhead and storage costs of this method.

By analysing Lare2D for two small scale runs (two and four cores) we can predict memory consumption of larger scale runs with a high degree of accuracy

(1% error at 64 cores). We also demonstrate how multiple models can be used to track the scaling behaviour of different phases of application execution.

We demonstrate the construction of a more complex model with the Chimaera application, taking ghost cells and processor decompositions into account. We were able to predict memory HWM to within 5% error on up to 512 cores, on the same problem, and 13% error for a smaller problem. Analysis of the growth factors attributed to MPI, allowed us to derive a new model with less than 9% error across the two problem classes.

The importance of these models, and the ability to predict memory consumption at scale, is in scheduling and potentially procurement. Knowing how much memory a certain problem size will require allows the user to schedule jobs on the correct core count, with sufficient memory resources available. A similar technique can be used to estimate machine memory requirements at the point of procurement, by predicting the memory consumption of either aspirational problem sizes or existing job sizes at increased core counts.

Another use of these models, presented in this chapter, is the evaluation of memory savings afforded by optimisation techniques ahead of implementation. We adapt our existing models to simulate the development of new runtime configurations of Chimaera to speculate on the memory savings afforded by both 3D processor decompositions and hybrid parallelism. We estimate that there could be a potential memory saving using a 3D processor decomposition of up to 18% over the equivalent 2D decomposition. Additionally we predict that using socket-level OpenMP with a 2D processor decomposition could theoretically reduce memory consumption by up to 75% when using oct-core processors.

Such investigations can provide a high level benefit analysis of the technique, from a memory perspective, ahead of time. The capabilities of WMTools mean this information can guide development efforts to save time and money by focusing on only those techniques which are likely to afford significant memory reductions.

CHAPTER 8
Conclusion

In this thesis we have developed a methodology for memory consumption data collection and analysis, and detailed the associated implementation of our memory consumption analysis suite WMTools. Using our memory analysis techniques we have been able to investigate the effects of consumption, and memory scalability, from the system workload level to the application level.

We have illustrated how to analyse existing applications and problems to identify memory consumption in both temporal and functional breakdowns. We demonstrate how to identify, and compare, artefacts in consumption across executions of different scale, in terms of both core counts and problem size. Combining this analysis we present a methodology to predict memory consumption, through automatically constructed memory models based on allocation size analysis.

Our research goal was to determine to what extent non-intrusive profiling methods be used to analyse application memory consumption? In this thesis we have not only shown the importance of memory consumption analysis, but developed a tool chain to provide low-level non-intrusive memory analysis. We have shown how these tools and techniques can be used to analyse individual memory allocations in parallel applications with minimal overheads. We have shown how this collected data can be used to evaluate memory scalability, identify memory consumption issues and even make predictions of potential application and problem set memory consumption. As such we feel like we have suitably answered the initial research question, and substantially contributed to the capabilities of the HPC community.

Whilst this research has been focused on HPC, and the benefits exhibited for a number of different applications there are a number of different applications to the methodologies and limitations to the techniques. One of the biggest limitations to our methodology is the risk of data size expansion. To achieve both fast and fine-grained analysis we trade off the storage of vast quantities of data, which must be post processed. This volume of data could be a prohibitive factor for especially large, or long, jobs. In such scenarios other techniques may be more suitable.

8.1 Contributions

This thesis presents the following novel contributions to the domain of memory consumption analysis:

- We have presented a method of collecting memory allocation data through function interposition, which enables the profiling of large scale parallel applications without the need for recompilation. By employing a method of data compression we store all allocation data to file, unlike other tools which discard much of this valuable information. Offline access to this information allows for a much greater level of analysis than is otherwise available. We have developed a suite of applications, WMTools, to implement this data collection and analysis methodology. This suite provides: HWM, temporal, functional and comparative analysis to gain a wide understanding of application memory behaviour. We have shown that WMTools provides increased facility for data analysis within a lightweight framework and demonstrated overheads comparable with other lightweight tools.

- By studying the HWM behaviour of a selection of applications, as they are strong scaled, we evaluated how reductions in memory-per-core ratios would affect workload runtime. Using the Maui scheduler we simulated the execution of artificial workloads of mixed science applications where strong scaling is used to reduce memory consumption to below the simulated memory-per-core ratio. From this analysis we emphasise the importance of efficient memory scalability in making applications more amenable to memory-per-core reductions, without associated runtime implications. Further, we looked into how two different programming paradigms (processor decompositions and hybrid parallelism) can be used to influence memory scalability. We demonstrated the magnitude of the available savings afforded by these techniques through the benchmark application SNAP.

- Using WMTools we investigated one of the more complex memory scaling nuances, MPI memory consumption. We demonstrated how our tools can be used to identify problems with existing OpenMPI implementations on different InfiniBand hardware. We were able to decompose memory consumption at HWM to attribute allocations to the MPI library, enabling us to track the growth in MPI consumption as the core count of our job is increased. Then we showed how different runtime configurations (both queue configurations and vendor-optimised libraries) could be used to address these MPI memory consumption issues. Our analysis identified that vendor-optimised libraries could be used to improve OpenMPI implementations, to provide efficient memory scalability.

- With an aim to address concerns over memory scalability, we demonstrated how analysis of current problems can be used to model application memory consumption, and thereby make predictions for large scale runs. We demonstrated how models generated from 'small scale' runs can be used to estimate, with high accuracy, the memory consumption of the application at different scales, in terms of either core count or problem size. This type of modelling can be of further use to procurement to estimate memory requirements for new machines, and identify any issues which may not have presented at small scale. Additionally we showed how these modes can be used to make conjectures about implementation changes, and their associated memory savings.

8.1.1 Beneficiaries

The tools and methodologies are designed to provide assistance to a wider range of HPC professionals.

At a high level it is crucial for system administrators to understand node level memory usage, to assist them with the management of supercomputers. Such information can be used to diagnose node failures and provide utilisation statistics.

This level of understanding is also required for requirements gathering for machine procurement. In such circumstance the needs of the applications are crucial to minimise expenditure. Further, the modelling techniques presented in Chapter 7 could be used to make predictions of application memory usage on new platforms providing higher core counts, but potentially with lower memory.

At the low level this information is crucial for application developers and code custodians. Understanding memory consumption can assist with debugging and the development of performance enhancements. Tools to analyse low-level memory consumption behaviour are crucial for the development and maintenance of high performance code, and ensuring the productivity of developers.

8.2 Limitations

By understanding the memory requirements of applications at different scale, we have achieved two main goals: the justification of memory-per-core ratios during procurement, and enabling the analysis to drive memory reductions – through both runtime configuration and application redesign.

Whilst the benefit of these goals is clear, there are certain limitations to our research, which we will address in this section.

8.2.1 Data Volume

One of the aspects that set WMTools apart from other memory analysis tools is the preservation of data. We have already discussed the motivation and benefits of this design decision in Section 4, but there are some limitations to this method.

The load placed on the network and I/O infrastructure can have a detrimental impact on the traced application, and also the supercomputer as a whole. Although we have demonstrated that this method is viable at current scale (up to a few thousand cores) we have not explored the true implications of large scale tracing. Our benchmarking applications have been constrained to application runs of up to an hour, as this is sufficient to gain adequate information, but this may not be an accurate representation of full scale production application runs. The data generated by an application run lasting upwards of a few days would be likely to prohibit the use of such profiling methods as have been presented in this thesis. Additionally the computational power to analyse the results would be likely to reflect the scale of the original job.

There are a number of potential solutions, which have not been explored in this thesis, that we will touch upon in Section 8.3.2.

Further, a very recent paper describes the process of using I/O forwarding techniques to improve performance of trace file storage for event-based parallel profiling tools [54]. Many of these techniques may be applicable to WMTools to increase the viability of memory tracing at increased scale.

8.2.2 Technologies

An original assumption of this body of research was the prevalence, and importance, of POSIX-based languages in HPC. Such an assumption is based on an understanding of current HPC applications, but no such assumptions can be made about future HPC applications.

Whilst legacy code bases are still the body of scientific workloads in many institutions there are a number of emerging programming languages and extensions which are not supported by the tools developed in this thesis.

The issue can be divided into two problems: the support for non heap based memory allocations, such as those on GPUs and other accelerators / co-processors, and the inability to capture heap based allocations in different programming languages (e.g Java).

The rise of accelerators has introduced another memory constrained device, where active memory management is required by the user. For devices where there is still a concept of host memory management calls the methodologies presented here for data collection are still applicable, but for devices with support for internal memory management we must look to alternative collection techniques. Assuming these techniques can generate sufficient information the analysis methods presented in WMTools are still valid, though an appreciation of allocation context would be required.

With regard to alternative programming languages, we have briefly discussed the use of OpenMP as a supplementary source of parallelism when used with MPI, and we have provided basic support within WMTools. We have not considered languages such as Java, as the use of a virtual machine facilitates a very different approach to instrumentation and tracing, and there exists an increasingly large body of research already addressing the issue. Additionally Java has never been a prevalent programming language in large scale HPC, due to the importance of performance. Java is not designed to optimise data locality, or facilitate high-level optimisations, additionally the metadata associated with high-level objects, and garbage collection, results in increased memory consumption.

Lack of support for emerging languages, which help expose parallelism, is also a limitation of our tools. Languages such as Charm++ [98] and Intel Cilk Plus [58] present an extension of C and C++ languages, with additional library calls to enable parallelism. Such a layout makes it hard to relate memory usage back to the underlying application, and so new data capture and analysis techniques may be required.

8.3 Future Work

Whilst this thesis presents a self-contained and complete body of research there are a number of potential extensions, which we were unable to address but believe would add further value to the field.

8.3.1 Static Source Code Analysis

The memory modelling work presented in Chapter 7 is based upon understanding relationships between allocation sizes in successive executions of an application. This technique is highly automated in its ability to generate models based on a few predictable parameters, but this does not allow for the application of domain knowledge.

During the analysis phase we know the source code location of all of the allocations, and their associated call stacks, and we believe we could use this information to further advance the models. Using static source code analysis it should be feasible to relate allocation sizes to runtime variables, whose origin can then be traced back through the application. We believe this analysis could then directly link allocation sizes to application problem parameters, and potentially the input deck for that problem. Such an understanding of the relationship between input parameters and memory consumption would facilitate the construction of increasingly complex models, which capture much deeper relationships.

Not only would this advanced modelling increase accuracy of memory scalability studies, but it would facilitate a new type of analysis based on problem parameters to understand which parameters have the most dramatic effect on memory scalability. This in turn could help the development of new input decks, with tuned parameters, to solve problems in a more memory economical way.

8.3.2 Mixed Mode Data Collection

One of the limitations discussed in Section 8.2 is the volume of data, and profiling overhead, of using WMTools on large scale production runs. In such a scenario it might be desirable to limit the profiling phase to a specific region of code, or application execution.

The methodology employed by WMTrace requires that all allocations and deallocations are caught and interpreted, thus limiting opportunities to refine the scope of analysis to a specific regions. One technique which could be employed to minimise some data storage and runtime overheads is using mixed mode data collection.

In Section 3.2 we detailed how different data collection techniques have inherent performance characteristics. By mixing WMTrace's interposition method with a lighter collection technique, we could enable non-allocation based consumption recording (e.g sampling RSS) throughout, but with a specific region of high intensity allocation based analysis.

As discussed, the data collected in sampling is not entirely compatible with interposition data, as there are no allocation call stacks or allocation addresses. This makes the interposition phase more complex, as some deallocations will not be matched. What this method would allow is knowing roughly how much memory was used, in total, throughout execution, in addition to tracking the allocations within the detailed phase.

In addition to the mixed data collection mode the user would need some method of interacting with the WMTrace library, to instruct it to enter, and later to exit, the detailed analysis phase. This is likely to be achieved through API calls, either WMTrace specific, or something based on the existing PMPI profiling level calls.

This data collection technique would still facilitate the generation of graphs and models, though not with the same detail and confidence as is available with our current technique.

8.3.3 Model Prediction Validations

In Section 7.5 we presented predictions for memory savings arising from implementation changes in the Chimaera application. Our conjecture is based on the change in decomposition arising from the technique, and the associated reduction in ghost cells and deduplication of data. Whilst we are confident in this technique, we understand that techniques for enhancing performance will increase memory consumption, and the savings may not be of quite the same magnitude as our predictions for a performant implementation.

We feel that, based on the predictions, Chimaera would deeply benefit from experimentation with these techniques; for that reason we would like to develop an implementation which takes these techniques into account. From this we will be able to validate our model predictions, and hopefully present some substantial memory savings.

8.3.4 Power Consumption

In Chapter 2.1.3 we discussed the importance of power consumption, in emerging technology. Whilst the research presented in this thesis relates to memory consumption, there are clear parallels between the analytical techniques required to relate the two topics. Understanding how much power an application is consuming, and where, is likely to become a crucial form of application analysis in the next few years.

Not only can the analysis methodologies in this thesis be adapted to understand application power consumption but the memory consumption analysis alone can be related to power consumption. As the power costs of data storage, and more crucially movement, begin to dominate overall application power consumption understanding the memory profile of applications can be used as auxiliary analysis to drive power reductions.

Bibliography

[1] K. Abe, M. P. Tendulkar, J. R. Jameson, P. B. Griffin, K. Nomura, S. Fujita, and Y. Nishi. Ultra-High Bandwidth Memory with 3D-Stacked Emerging Memory Cells. In *Integrated Circuit Design and Technology and Tutorial, 2008. ICICDT 2008. IEEE International Conference on*, pages 203–206, 2008.

[2] AnandTech. NVIDIA's GPU Technology Conference 2013 Keynote Live Blog. `http://www.anandtech.com/show/6842/nvidias-gpu-technology-conference-2013-keynote-live-blog`.

[3] T. D. Arber, A. W. Longbottom, C. L. Gerrard, and A. M. Milne. A Staggered Grid, Lagrangian–Eulerian Remap Code for 3-D MHD Simulations. *Multiple values selected*, 171(1):151–181, July 2001.

[4] AWE plc. Ichnaea (PMTM). `http://sourceforge.net/projects/ichnaea/`.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, and R. S. Schreiber. The NAS Parallel Benchmarks Summary and Preliminary Results. *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 158–165, 1991.

[6] R. S. Baker and R. E. Alcouffe. DANTSYS/MPI–A System for 3-D Deterministic Transport on Parallel Architectures. *Three-dimensional deterministic radiation transport computer programs*, pages 1–17, 1997.

[7] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. Träff. MPI on a Million Processors. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30, 2009.

[8] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient Management of Parallelism in Object Oriented Numerical Software Libraries. In *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[9] C. Batten, A. Joshi, J. Orcutt, A. Khilo, B. Moss, C. W. Holzwarth, M. A. Popovic, H. Li, H. I. Smith, J. L. Hoyt, F. X. Kartner, R. J. Ram, V. Stojanovic, and K. Asanovic. Building Many-Core Processor-to-DRAM Networks with Monolithic CMOS Silicon Photonics. *Micro, IEEE*, 29(4):8–21.

[10] S. Beamer, C. Sun, Y.-J. Kwon, A. Joshi, C. Batten, V. Stojanovic, and K. Asanovic. Re-Architecting DRAM Memory Systems with Monolithically Integrated Silicon Photonics. *ACM SIGARCH Computer Architecture News*, 38(3):129–140, 2010.

[11] J. Beckett. Memory Performance Guidelinesfor Dell PowerEdge 12thGeneration Servers . `http://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/12g-memory-performance-guide.pdf`.

[12] J. Bell, M. Berger, J. Saltzman, and M. Welcome. Three-Dimensional Adaptive Mesh Refinement for Hyperbolic Conservation Laws. *SIAM Journal on Scientific Computing*, 15(1), Jan. 1994.

[13] M. J. Berger and P. Colella. Local Adaptive Mesh Refinement for Shock Hydrodynamics. *Journal of Computational Physics*, 82(1), May 1989.

[14] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead. Technical report, 2008.

[15] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-Based File Backup. *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1–9, 2009.

[16] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric Prediction of Heap Memory Requirements. In *ISMM '08: Proceedings of the 7th international symposium on Memory management*. ACM Request Permissions, June 2008.

[17] D. L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Sept. 2004.

[18] D. W. Chang, N. S. Kim, and M. J. Schulte. Analyzing the Performance and Energy Impact of 3D Memory Integration on Embedded DSPs. In

Embedded Computer Systems (SAMOS), 2011 International Conference on, pages 303–310, 2011.

[19] M.-F. Chang, P.-F. Chiu, W.-C. Wu, C.-H. Chuang, and S.-S. Sheu. Challenges and Trends in Low-Power 3D Die-Stacked IC Designs Using RAM, Memristor Logic, and Resistive Memory (ReRAM). In *ASIC (ASICON), 2011 IEEE 9th International Conference on*, pages 299–302, 2011.

[20] M.-F. Chang, C.-S. Lin, W.-C. Wu, M.-P. Chen, Y.-H. Chen, Z.-H. Lin, S.-S. Sheu, T.-K. Ku, C.-H. Lin, and H. Yamauchi. A High Layer Scalability TSV-Based 3D-SRAM With Semi-Master-Slave Structure and Self-Timed Differential-TSV for High-Performance Universal-Memory-Capacity-Platforms. *Solid-State Circuits, IEEE Journal of*, 48(6):1521–1529, 2013.

[21] L. Chen, T. Lu, Y. Wang, M. Chen, Y. Ruan, Z. Cui, Y. Huang, M. Chen, J. Zhang, and Y. Bao. MIMS: Towards a Message Interface Based Memory System. *arXiv.org*, Jan. 2013.

[22] Y. Chen, X. Cui, and H. Mei. Large-Scale FFT on GPU Clusters. In *ICS '10: Proceedings of the 24th ACM International Conference on Supercomputing*. ACM Request Permissions, June 2010.

[23] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *Micro, IEEE*, 30(2):16–29, 2010.

[24] E. Cooper-Balis, P. Rosenfeld, and B. Jacob. Buffer-on-Board Memory Systems. *SIGARCH Comput. Archit. News*, 40(3):392–403, 2012.

[25] Crucial. DDR3 Prices. `http://www.crucial.com/store/listmodule/DDR3/list.html`.

[26] T. W. Curry. Profiling and Tracing Dynamic Library Usage via Interposition. *USENIX Summer 1994 Technical Conference*, pages 267–278, 1994.

[27] M. Daneshtalab, M. Ebrahimi, P. Liljeberg, J. Plosila, and H. Tenhunen. Memory-Efficient Logic Layer Communication Platform for 3D-Stacked Memory-on-Processor Architectures. In *3D Systems Integration Conference (3DIC), 2011 IEEE International*, pages 1–8, 2012.

[28] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz. CPU DB. *Communications of the ACM*, 55(4):55, Apr. 2012.

[29] J. A. Davis, G. R. Mudalige, S. D. Hammond, J. A. Herdman, I. Miller, and S. A. Jarvis. Predictive Analysis of a Hydrodynamics Application on Large-Scale CMP Clusters. *Computer Science - Research and Development*, 26(3-4), June 2011.

[30] C. T. Delistavrou and K. G. Margaritis. Survey of Software Environments for Parallel Distributed Processing: Parallel Programming Education on Real Life Target Systems Using Production Oriented Software Tools. In *2010 14th Panhellenic Conference on Informatics (PCI)*. IEEE.

[31] Q. Deng, L. Ramos, R. Bianchini, D. Meisner, and T. Wenisch. Active Low-Power Modes for Main Memory with MemScale. *Micro, IEEE*, 32(3):60–69, 2012.

[32] B. Diniz, D. Guedes, W. Meira, Jr, and R. Bianchini. Limiting the Power Consumption of Main Memory. *ACM SIGARCH Computer Architecture News*, 35(2):290–301, 2007.

[33] J. Draper, J. Chame, M. Hall, C. Steele, T. Barrett, J. LaCoss, J. Granacki, J. Shin, C. Chen, and C. W. Kang. The Architecture of the DIVA Processing-in-Memory Chip. *Proceedings of the 16th international conference on Supercomputing*, pages 14–25, 2002.

[34] R. Drost, C. Forrest, B. Guenin, R. Ho, A. V. Krishnamoorthy, D. Cohen, J. E. Cunningham, B. Tourancheau, A. Zingher, and A. Chow. Challenges in Building a Flat-Bandwidth Memory Hierarchy for a Large-Scale Computer with Proximity Communication. *Proceedings of the 13th Symposium on High Performance Interconnects*, pages 13–22, 2005.

[35] R. Falgout and U. Yang. hypre: a Library of High Performance Preconditioners. *Computational Science—ICCS 2002*, pages 632–641, 2002.

[36] Z. Fang, L. Zhang, J. B. Carter, S. A. McKee, A. Ibrahim, M. A. Parker, and X. Jiang. Active Memory Controller. *Journal of Supercomputing*, 62(1):510–549, 2012.

[37] M. Farreras, T. Cortes, J. Labarta, and G. Almasi. Scaling MPI to Short-Memory MPPs Such as BG/L. In *ICS '06: Proceedings of the 20th annual international conference on Supercomputing*. ACM Request Permissions, June 2006.

[38] K. B. Ferreira, R. Riesen, D. Arnold, D. Ibtesham, and R. Brightwell. The Viability of Using Compression to Decrease Message Log Sizes. In

Euro-Par'12: Proceedings of the 18th international conference on Parallel processing workshops. Springer-Verlag, Aug. 2012.

[39] I. Finocchi. Software Streams: Big Data Challenges in Dynamic Program Analysis. *The Nature of Computation Logic*, 2013.

[40] N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-Overhead Call Path Profiling of Unmodified, Optimized Code. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. ACM Request Permissions, June 2005.

[41] Fujitsu. Memory Performance of Xeon E5-2600/4600 (Sandy Bridge-Ep) Based Systems. `http://globalsp.ts.fujitsu.com/dmsp/Publications/public/wp-sandy-bridge-ep-memory-performance-ww-en.pdf`, Mar. 2012.

[42] A. Gainaru, F. Cappello, and W. Kramer. Taming of the Shrew: Modeling the Normal and Faulty Behaviour of Large-Scale HPC Systems. In *2012 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1168–1179. IEEE.

[43] B. Ganesh, A. Jaleel, D. Wang, and B. Jacob. Fully-Buffered DIMM Memory Architectures: Understanding Mechanisms, Overheads and Scaling. *Proceedings of the 13th International Symposium on High Performance Computer Architecture*, pages 109–120, 2007.

[44] J. Gilchrist and A. Cuhadar. Parallel Lossless Data Compression Using the Burrows-Wheeler Transform. *International Journal of Web and Grid Services*, 4(1), May 2008.

[45] F. G. Gustavson and T. Swirszcz. In-Place Transposition of Rectangular Matrices. In *PARA'06: Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*. Springer-Verlag, June 2006.

[46] S. D. Hammond, G. R. Mudalige, J. A. Smith, S. A. Jarvis, J. A. Herdman, and A. Vadgama. WARPP - A Toolkit for Simulating High-Performance Parallel Scientific Codes. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 19:1–19:10, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[47] W. Hassanein, J. Fortes, and R. Eigenmann. Data Forwarding Through in-Memory Precomputation Threads. *Proceedings of the 18th Annual International Conference on Supercomputing*, pages 207–216, 2004.

[48] D. Hepkin. Active Memory Expansion. Technical report, Systems and Technology Group, IBM, Feb. 2010.

[49] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving Performance via Mini-applications. Technical Report SAND2009-5574, Sept. 2009.

[50] Hewlett-Packard. *DDR3 Memory Technology*, Apr. 2010.

[51] M. Hofmann and S. Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Request Permissions, Jan. 2003.

[52] HP. Memory Technology in HP Z Workstations. `http://h20331.www2.hp.com/Hpsub/downloads/HP_WS_MemoryTechnology_041812.pdf`, Apr. 2012.

[53] IBM. IBM Power Systems. `http://www-05.ibm.com/cz/events/febannouncement2012/pdf/power_architecture.pdf`, Feb. 2013.

[54] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knupfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole. Optimizing I/O Forwarding Techniques for Extreme-Scale Event Tracing. *Cluster Computing*, June 2013.

[55] Intel. Intel ARK. `http://ark.intel.com/`.

[56] Intel. Accelerating Silicon Design with theIntel® Xeon® Processor E7-4800 Product Family. `http://www.intel.ie/content/dam/www/public/us/en/documents/technology-briefs/intel-it-xeon-e7-4800-accelerating-silicon-design-brief.pdf`, Apr. 2011.

[57] Intel. Intel® 7500/7510/7512 Scalable Memory Buffer. `http://www.intel.co.uk/content/dam/doc/datasheet/7500-7510-7512-scalable-memory-buffer-datasheet.pdf`, Apr. 2011.

[58] INTELCorporation. Intel Cilk Plus. `http://software.intel.com/en-us/intel-cilk-plus`, 2011.

[59] D. B. Jackson, H. L. Jackson, and Q. O. Snell. Simulation Based HPC Workload Analysis. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, 2001.

[60] J. Jeddeloh and B. Keeth. Hybrid Memory Cube New DRAM Architecture Increases Density and Performance. *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88, 2012.

[61] JEDEC. JEDEC STANDARD: JESD79-4. Technical report, Sept. 2012.

[62] E. Karrels and E. Lusk. Performance Analysis of MPI Programs. In J. J. Dongarra and B. Tourancheau, editors, *Proceedings of the Workshop on Environments and Tools For Parallel Scientific Computing*, pages 195–200. SIAM, 1994.

[63] J.-M. Kim, J.-K. Du, J.-R. Yuk, and J.-G. Yook. Noise Coupling Analysis of the High Speed Memory Module. *Antennas and Propagation Society International Symposium*, pages 3948–3951, 2007.

[64] P. M. Kogge. The EXECUBE Approach to Massively Parallel Processing. In *Proceedings of the International Confonference of Parallel Processing*, 1994.

[65] P. M. Kogge and T. J. Dysart. Using the Top500 to Trace and Project Technology and Architecture Trends. *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 28, 2011.

[66] M. J. Koop, T. Jones, and D. K. Panda. Reducing Connection Memory Requirements of MPI for InfiniBand Clusters: A Message Coalescing Approach. In *Proceedings of the 7th IEEE International Symposium on Cluster Computing and the Grid (CCGRID'07)*, pages 495–504, May 2007.

[67] M. J. Koop, S. Sur, Q. Gao, and D. K. Panda. High Performance MPI Design using Unreliable Datagram for Ultra-scale InfiniBand Clusters. In *Proceedings of the 2007 IEEE/ACM International Conference on Supercomputing*, pages 180–189, New York, NY, USA, May 2007. ACM.

[68] C. Kügeler, M. Meier, R. Rosezin, S. Gilles, and R. Waser. High Density 3D Memory Architecture Based on the Resistive Switching Effect. *Solid State Electronics*, 53(12):1287–1292, Dec. 2009.

[69] Lawrence Livermore National Laboratory. SILO. `https://wci.llnl.gov/codes/silo/`.

[70] Lawrence Livermore National Laboratory. VisIt. `https://wci.llnl.gov/codes/visit/`.

[71] S. Levy, K. B. Ferreira, P. G. Bridges, A. P. Thompson, and C. Trott. An Examination of Content Similarity Within the Memory of HPC Applications. Technical Report SAND2013-0055, Jan. 2013.

[72] H. Li, D. Groep, J. Templon, and L. Wolters. Predicting Job Start Times on Clusters. *Cluster Computing and the Grid, 2004. CCGrid 2004. IEEE International Symposium on*, pages 301–308, 2004.

[73] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated Memory for Expansion and Sharing in Blade Servers. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. ACM Request Permissions, June 2009.

[74] J. Lin, H. Zheng, Z. Zhu, H. David, and Z. Zhang. *Thermal Modeling and Management of DRAM Memory Systems*, volume 35. ACM, 2007.

[75] J. Liu and others. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *Proceedings of the 2003 ACM/IEEE International Conference on Supercomputing*, pages 58–71, New York, NY, USA, 2003. ACM.

[76] G. H. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 453–464, 2008.

[77] I. Loi and L. Benini. An Efficient Distributed Memory Interface for Many-Core Platform with 3D Stacked DRAM. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 99–104, 2010.

[78] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer. End-User Tools for Application Performance Analysis Using Hardware Counters. *International Conference on Parallel and Distributed Computing Systems*, pages 8–10, 2001.

[79] S.-L. Lu, T. Karnik, G. Srinivasa, K.-Y. Chao, D. Carmean, and J. Held. Scaling the "Memory Wall". In *ICCAD '12: Proceedings of the International Conference on Computer-Aided Design*. ACM Request Permissions, Nov. 2012.

[80] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[81] Q. Meng, M. Berzins, and J. Schmidt. Using Hybrid Parallelism to Improve Memory Use in the Uintah Framework. In *TG '11: Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. ACM Request Permissions, July 2011.

[82] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. TOP500 Supercomputer Sites. `http://www.top500.org`.

[83] H. W. Meuer and H. Gietl. Supercomputers–Prestige Objects or Crucial Tools for Science and Industry? *Praxis der Informationsverarbeitung und Kommunikation*, pages 1–12, 2012.

[84] Micron. 64GB 1.35V DDR3L SDRAM LRDIMM. `http://www.micron.com/~/media/Documents/Products/Data%20Sheet/Modules/kszq144c8gx72lz.pdf`, May 2013.

[85] MicronTechnology Inc. DDR3 to DDR4. `http://www.micron.com/products/dram/ddr3-to-ddr4`.

[86] F. Mietke, R. Rex, R. Baumgartl, T. Mehlan, T. Hoefler, and W. Rehm. Analysis of the Memory Registration Process in the Mellanox Infiniband Software Stack. In *Proceedings of the 12th international conference on Parallel Processing*, pages 124–133, Berlin, Heidelberg, 2006. Springer-Verlag.

[87] B. Mohr, A. D. Malony, S. Shende, and F. Wolf. Design and Prototype of a Performance Tool Interface for OpenMP. *Journal of Supercomputing*, 23(1):105–128, 2002.

[88] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *2009 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 261–270. IEEE.

[89] P. Mosalikanti, C. Mozak, and N. Kurd. High performance DDR architecture in Intel® Core^{TM} processors using 32nm CMOS high-K metal-gate process. *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*, pages 1–4, 2011.

[90] D. Mosberger, A. Sharma, J. F. A. Paulino, B. Sumner, H. Boehm, M. Delahaye, and L. Tuura. libunwind. 2011.

[91] G. R. Mudalige, M. Vernon, and S. A. Jarvis. A Plug-and-Play Model for Evaluating Wavefront Computations on Parallel Architectures. *Parallel*

and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on, pages 1–14, 2008.

[92] T. Mytkowicz, D. Coughlin, and A. Diwan. Inferred Call Path Profiling. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. ACM Request Permissions, Oct. 2009.

[93] N. Nethercote and J. Seward. How to Shadow Every Byte of Memory used by a Program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 65–74, New York, NY, USA, 2007. ACM.

[94] N. Nethercote and J. Seward. Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming language design and implementation*, pages 89–100, New York, NY, USA, 2007. ACM.

[95] NETLIST. Effective Data Rates on Fully Populated 3DPC Servers. `http://www.netlist.com/files/9113/4548/7143/20120814_NL_White_Paper_Effective_Data_Rates_on_3DPC_Servers.pdf`, Aug. 2012.

[96] NVIDIA Corporation. Tesla Kepler Product Overview. `http://www.nvidia.com/content/tesla/pdf/Tesla-KSeries-Overview-LR.pdf`, Oct. 2012.

[97] OpenFabrics Alliance. OFED Overview. `https://www.openfabrics.org/resources/ofed-for-linux-ofed-for-windows/ofed-overview.html`.

[98] Parallel Programming Library, University of Illinois. Charm++. `http://charm.cs.uiuc.edu/research/charm/`.

[99] D. Parkinson. Parallel Efficiency Can Be Greater Than Unity. *Parallel Computing*, 3(3), July 1986.

[100] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *Micro, IEEE*, 17(2):34–44, 1997.

[101] O. F. J. Perks. WMTools - GitHub. `https://github.com/Warwick-PCAV/WMTools`.

[102] O. F. J. Perks, D. A. Beckingsale, A. S. Dawes, J. A. Herdman, C. Mazauric, and S. A. Jarvis. Analysing the Influence of InfiniBand Choice on OpenMPI Memory Consumption. *International Workshop on High Performance Interconnection Networks (HPIN)*, pages 1–8, Mar. 2013.

[103] O. F. J. Perks, D. A. Beckingsale, S. D. Hammond, I. Miller, J. A. Herdman, A. Vadgama, A. H. Bhalerao, L. He, and S. A. Jarvis. Towards Automated Memory Model Generation Via Event Tracing. *The Computer Journal*, 56(2):156–174, June 2012.

[104] O. F. J. Perks, R. F. Bird, D. A. Beckingsale, and S. A. Jarvis. Exploiting Spatiotemporal Locality for Fast Call Stack Traversal. *Workshop on High-performance Infrastructure for Scalable Tools (WHIST)*, pages 1–8, June 2012.

[105] O. F. J. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. Should We Worry About Memory Loss? *SIGMETRICS Performance Evaluation Review*, 38(4), Mar. 2011.

[106] O. F. J. Perks, S. D. Hammond, S. J. Pennycook, and S. A. Jarvis. WMTools - Assessing Parallel Application Memory Utilisation at Scale. *Proceedings of the European Perfomance Evaluation Workshop (EPEW)*, 6977:148–162, Oct. 2011.

[107] QLogic. The Impact of InfiniBand Architecture on CPU Utilization. `http://qlogic.com/Resources/Documents/WhitePapers/InfiniBand/Impact_IB_Architecture_on_CPU_Utilization.pdf`, July 2011.

[108] R. Rabenseifner. Automatic Profiling of MPI Applications with Hardware Performance Counters. In *Proceedings of the 6th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Springer-Verlag, Sept. 1999.

[109] A. Rodrigues, R. Murphy, R. Brightwell, and K. D. Underwood. Enhancing NIC Performance for MPI Using Processing-in-Memory. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE Computer Society, 2005.

[110] M. Serrano and X. Zhuang. Building Approximate Calling Context from Partial Call Traces. In *CGO '09: Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, pages 221–230. IEEE Computer Society, Mar. 2009.

[111] G. Shainer, T. Wilde, P. Lui, T. Liu, M. Kagan, M. Dubman, Y. Shahar, R. Graham, P. Shamis, and S. Poole. The Co-Design Architecture for Exascale Systems, a Novel Approach for Scalable Designs. *Computer Science - Research and Development*, May 2012.

[112] N. Shida, S. Sumimoto, and A. Uno. MPI Library and Low-Level Communication on the K Computer. *FUJITSU Scientific & Technical Journal*, 2012.

[113] G. Shipman, R. Brightwell, B. Barrett, J. Squyres, and G. Bloch. Investigations on InfiniBand: Efficient Network Buffer Utilization at Scale. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 178–186, 2007.

[114] G. Shipman, S. Poole, P. Shamis, and I. Rabinovitz. X-SRQ-Improving Scalability and Performance of Multi-Core InfiniBand Clusters. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 33–42, 2008.

[115] G. M. Shipman, T. S. Woodall, R. L. Graham, A. B. Maccabe, and P. G. Bridges. InfiniBand Scalability in OpenMPI. *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, 2006.

[116] G. Singh and E. Deelman. The Interplay of Resource Provisioning and Workflow Optimization in Scientific Applications. *Concurrency and Computation: Practice & Experience*, 23(16), Nov. 2011.

[117] J. P. Stevenson, A. Firoozshahian, A. Solomatnikov, M. Horowitz, and D. Cheriton. Sparse Matrix-Vector Multiply on the HICAMP Architecture. In *ICS '12: Proceedings of the 26th ACM international conference on Supercomputing*. ACM Request Permissions, June 2012.

[118] H. S. Stone. A Logic-in-Memory Computer. *Computers, IEEE Transactions on*, (1):73–78, 1970.

[119] W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang. Precise Calling Context Encoding. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 525–534. ACM Request Permissions, May 2010.

[120] Super Micro Computer, Inc. Memory Configuration Guide. `http://www.supermicro.com/support/resources/memory/X9_DP_memory_config_socket_R.pdf`, Mar. 2012.

[121] Z. Szebenyi, T. Gamblin, M. Schulz, B. R. d. Supinski, F. Wolf, and B. J. N. Wylie. Reconciling Sampling and Direct Instrumentation for Unintrusive Call-Path Profiling of MPI Programs. In *IPDPS '11: Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, pages 640–651, Washington, DC, USA, May 2011. IEEE Computer Society.

[122] R. B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M. E. Wazlowski, and P. M. Bland. IBM Memory Expansion Technology (MXT). *IBM Journal of Research and Development*, 45(2), Mar. 2001.

[123] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*. ACM, Dec. 2002.

[124] D. W. Walker. Characterizing the Parallel Performance of a LargeScale, ParticleinCell Plasma Simulation Code. *Concurrency: Practice and experience*, 2(4):257–288, 1990.

[125] J. Whaley. A Portable Sampling-Based Profiler for Java Virtual Machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*. ACM, June 2000.

[126] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Hybrid Cache Architecture with Disparate Memory Technologies. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*. ACM Request Permissions, June 2009.

[127] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, and Y. Xie. Design Exploration of Hybrid Caches with Disparate Memory Technologies. *Transactions on Architecture and Code Optimization (TACO*, 7(3), Dec. 2010.

[128] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[129] Y. Xie. Future Memory and Interconnect Technologies. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013*, pages 964–969, 2013.

[130] J. Yoo, S. Yoo, and K. Choi. Active Memory Processor for Network-on-Chip-Based Architecture. *Computers, IEEE Transactions on*, 61(5):622–635, 2012.

[131] Q. Zhao, D. Bruening, and S. Amarasinghe. Umbra: Efficient and Scalable Memory Shadowing. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM Request Permissions, Apr. 2010.

[132] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. USENIX Association, Feb. 2008.

[133] zlib. zlib. `http://www.zlib.net`.

Appendices

APPENDIX A
Context: Architectures and Applications

In this chapter we discuss some of the factors affecting our benchmarking and testing environment.

To evaluate the performance and capability of tools and methodologies we need to utilise some representative HPC applications and execute them on HPC platforms. We have utilised a number of production HPC platforms, of differing size and age, to demonstrate tool capabilities at different scale. In some scenarios we utilise our tools to monitor the differences in hardware behaviour across similar systems.

For the benchmarking applications we make use of a mixture of scientific codes from different institutions, which exhibit different properties, to allow us to compare tool performance and capability in different situations. Whilst these applications are generally reduced form versions of production applications they exhibit many of the same characteristics but run faster.

A.1 Machines

	Cab	Hera	Minerva	Kay
Institution	LLNL	LLNL	Warwick	Bull
Processor	Intel Xeon	AMD Opteron	Intel Xeon	Intel Xeon
Processor Model	E5-2670	8356	X5650	E5-2680
Cores / Node	16	16	12	16
RAM / Node	32 GB	32 GB	24 GB	64 GB
Nodes	1,296	864	396	150
Network Vendor	QLogic	Mellanox	QLogic	Mellanox
Network Type	IB QDR	IB DDR	IB QDR	IB QDR

Table A.1: Computer system specifications

Table A.1 documents the machine specification of the various supercomputers

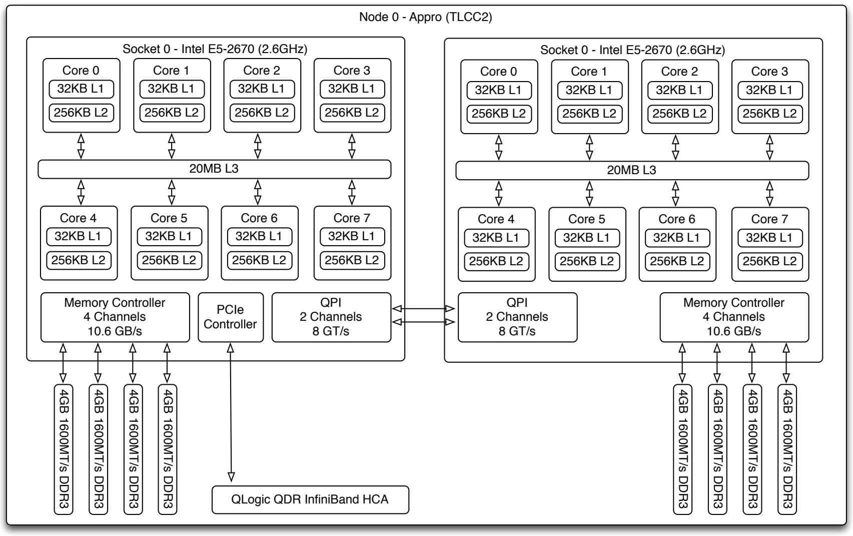


Figure A.1: Node level structure of Cab

utilised during this research. All computers displayed, with the exception of Kay, are production platforms and are supporting scientific research at academic and national research facilities. Kay represents a development platform at Bull, a supercomputer production company, and is utilised for application and hardware evaluation during the machine procurement process.

The specifications presented here represent a snapshot of the machine configuration at the time of utilisation. Whilst they may have changed since access, our results are based on these specifications.

A.1.1 Cab (LLNL)

The Cab platform at LLNL is part of the Tri-labs Linux Capacity Cluster 2 (TLCC2) project – an initiative to procure capacity supercomputers in terms of scalable units (SUs) using the buying power of the tri-labs (LLNL, LANL and SNL).

The platform is produced by Appro and Cab represents 8 SUs, utilising the Sandy Bridge generation of Intel processor, connected with a QLogic QDR

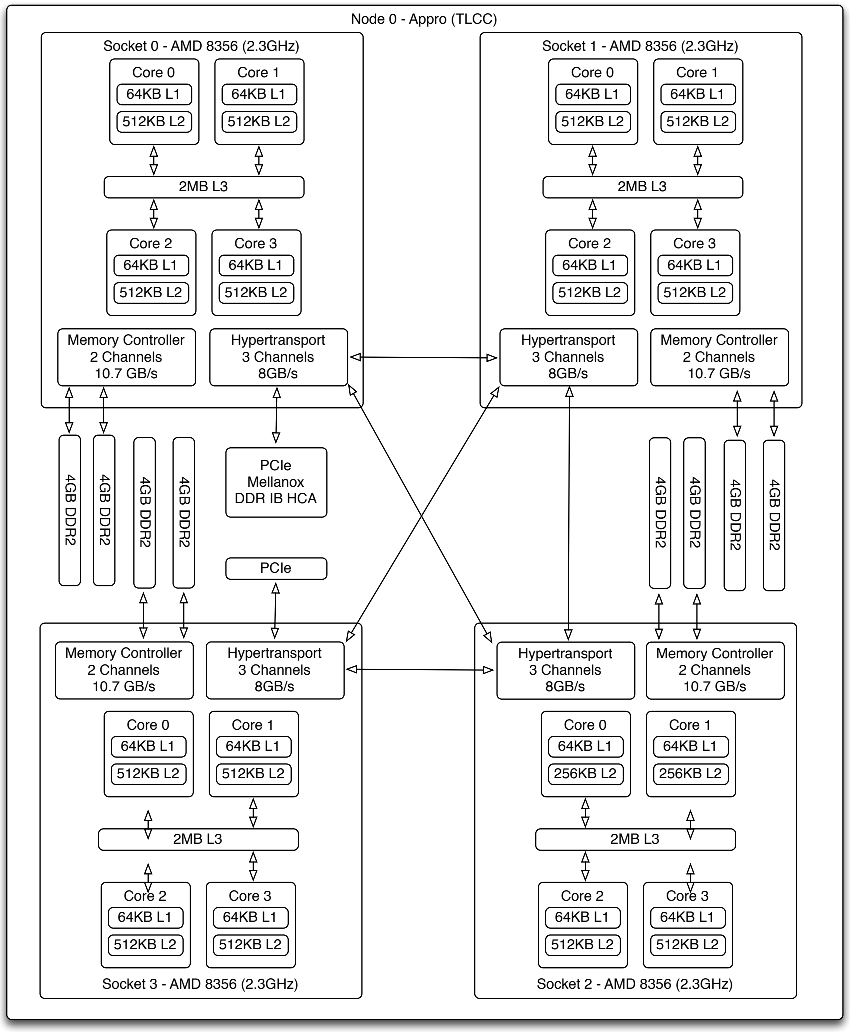


Figure A.2: Node level structure of Hera

InfiniBand. The structure of each node is presented in Figure A.1.

A.1.2 Hera (LLNL)

The Hera platform has been decommissioned since our initial research. Commissioned in 2008 the platform formed part of the initial TLCC procurement project, consisting of 6 SUs. Unlike the other platforms this is a quad-socket

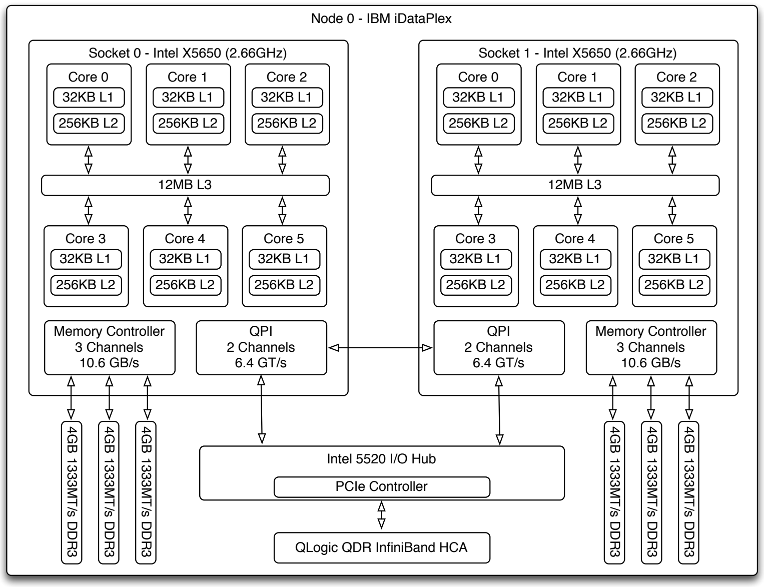


Figure A.3: Node level structure of Minerva

node, using AMD Barcelona generation quad-core chips.

Due to its age, the platform uses DDR InfiniBand making use of Mellanox HCAs.

From Figure A.2, we can see that each socket only supports three Hypertransport channels, meaning that there are insufficient channels for a fully connected system between the four sockets. As socket 0 and 3 are not directly connected, memory accesses between them require a two hop communication (via another socket) and the added latency may diminish HPC application performance.

A.1.3 Minerva (Warwick)

The Minerva platform (Figure A.3) at the Centre for Scientific Computing (CSC) supports the scientific computational workload of Warwick University. The computer was commissioned in 2011, to replace the Francesca supercomputer, and is based on the IBM iDataPlex platform.

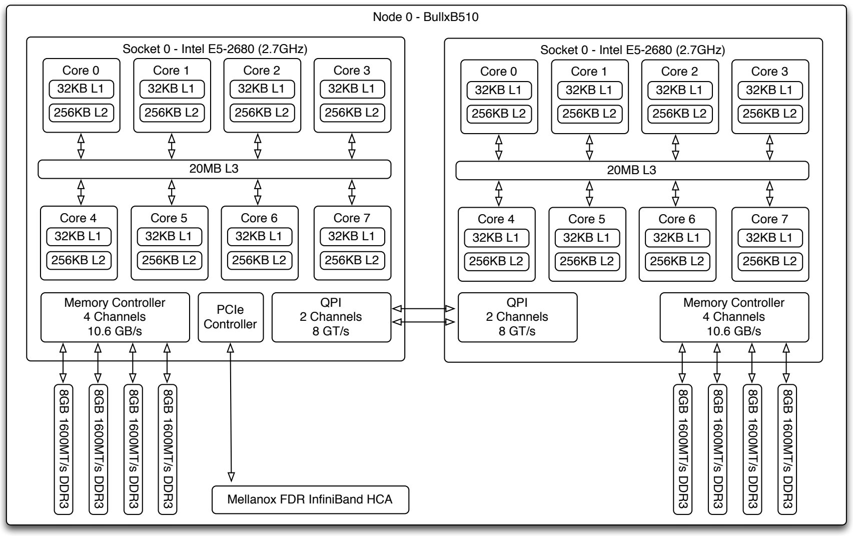


Figure A.4: Node level structure of Kay

Part of the platform is supported by the MidPlus consortium: the University of Warwick, the University of Birmingham, the University of Nottingham and Queen Mary, University London. A significant portion of the computational resource is reserved for MHD research.

A.1.4 Kay (Bull)

The Kay platform at Bull is a heterogeneous cluster comprised of multiple partitions of homogenous hardware. It is a primarily an internal benchmarking and research platform, so has a constantly evolving configuration. The partition utilised during this research is SNBEP64, consisting of 150 BullxB510 nodes, as described in Figure A.4. The platform, in this configuration, has since been disbanded and replaced with an Intel Xeon Ivy Bridge platform connected with FDR InfiniBand.

Although this figure indicates that the nodes utilise an FDR InfiniBand HCA, they are connected to a QDR backplane. Thus the operating speed of the network defaults to QDR performance.

A.2 Applications

Throughout this thesis we utilise a collection of benchmark applications and a class of small applications referred to as ‘mini-apps’. Such applications are designed to exhibit the computational behaviour of key algorithms, or processes, but in a simplistic and portable framework. Projects such as the Mantevo project at SNL provide a suite of mini-apps targeted at different scientific domains; they are designed to quickly evaluate hardware, both novel and traditional, and software methods [49].

Benchmark applications tend to differ from mini-apps in terms of both size and complexity, as they are designed to more accurately represent the computational needs of production grade applications. Benchmarks can play a key role in the procurement process, by evaluating a platform for both compatibility and performance. Such applications often contain reduced computational complexity and are accompanied with reduced problem sets, allowing for the fast turnaround of computational results during machine evaluation.

Both classes of application are suitable for analysis, specifically with respect to memory consumption, as they are designed to mimic the methods of larger codes and so will exhibit many of the same properties and artefacts.

A.2.1 Chimaera (AWE)

The Chimaera benchmark is a 3D particle transport code developed and maintained by AWE. It employs a wavefront design pattern, which executes a series of sweeps through the 3D data array. The purpose of the benchmark is the replication of operational behaviour of larger internal codes which occupy a considerable proportion of parallel runtime on the supercomputing facilities of AWE.

The code shares many similarities with the ubiquitous Sweep3D application developed by the LANL in the United States, but is considerably larger and more complex in its operation.

A.2.2 Orthrus (AWE)

Orthrus was initially developed by Dawes at AWE plc, to assess the parallel scalability of generic 3D implicitly solved linear diffusion problems. The application serves as a driver for the third-party linear solver libraries `PETSc` [8], from Argonne National Laboratory; and `hypre` [35], from LLNL. The application constructs a 3D sparse matrix and then drives the preconditioner-solvers provided by the two aforementioned libraries.

Orthrus forms part of the machine evaluation benchmark suite used to drive procurement decisions for AWE. Timing instrumentation is provided via the Ichnaea (PMTM) library [4].

A.2.3 POP (LANL)

The Parallel Ocean Program (POP) is a 3D ocean circulation model, solving equations for fluid motion on a sphere, using finite difference discretisation. POP forms part of the the Community Climate System Model, and as such it can be coupled with other climate simulators for more comprehensive modelling.

A.2.4 SNAP (LANL)

SNAP is a 3D SN proxy application, for the LANL neutron transport code PARTISN, designed to mimic memory requirements and communication patterns rather than physics. As such SNAP allows the configuration of a number of runtime parameters such as data cells, energy groups and angles, each of which can have a dramatic effect on both runtime and memory consumption.

SNAP also supports a level of hybrid parallelism with MPI and OpenMP, making it a suitable code with which to investigate memory effects.

A.2.5 Sweep3D (LANL)

Sweep3D was the precursor to the SNAP application, and shares many of the same characteristics both in terms of application and implementation.

A.2.6 NPB (NASA)

NASA maintains a benchmark suite of applications, referred to as the NAS Parallel Benchmark (NPB) suite [5]. There are many variants of the suite, where different programming languages or parallelisation paradigms are implemented. The Fortran variant of NPB makes heavy use of statically allocated arrays, as the runtime core count and problem are specified at compile time, making an efficient, but non-portable binary.

MG

MG is a multi grid solver utilising the Poisson solver. It is quite a memory intensive application despite having low heap usage, due to the use of static allocations discussed above.

A.2.7 LAMMPS (SNL)

LAMMPS (Large-scale Atomic/Molecular Massively Parallel Simulator) is a framework for classical molecular dynamics. It is designed for modelling systems of millions, and even billions, of particles on large scale HPC platforms. Whilst LAMMPS contains support for a wide range of particles and interaction models it also supports modification to allow the user to develop more custom behaviour.

Written in C++, LAMMPS has support for Fortran interfaces and is inherently designed as a parallel application using MPI, and even includes some GPU support.

A.2.8 MiniFE (SNL)

MiniFE is a proxy application, representing key functionality from the Sandia SIERRA suite of finite-element applications in a small and portable application. It is used to test programming languages and parallelisation models. It is an instructed implicit finite-element solver using a sparse linear system, constructed from steady-state conduction equations.

We present an evaluation of MiniFE's memory consumption characteristics in [106], where we investigate the effects of problem size and core count of temporal memory consumption.

A.2.9 phdMesh (SNL)

phdMesh is a finite element data structure library for parallel heterogeneous direct unstructured meshes, developed at SNL and initially included as part of the Mantevo benchmark suite, and is part of the Trilinos project.

Our usage of the code is based upon the 'gears' problem, which undertakes contact search on the unstructured grid. The use of an unstructured mesh makes it an interesting application to analyse in terms of memory consumption, as it is likely to exhibit a different profile to traditional structured mesh applications.

The version of phdMesh utilised in our research is written in C++, with a particularly high rate of object creation and destruction. This makes it a very interesting code to evaluate the performance of tracing tools with. In [106] we demonstrated that WMTools exhibited an application slowdown of up to 11.5× slowdown when profiles the code, but also that this behaviour is in line with other tools.

A.2.10 Lare2D (Warwick)

Lare2D is a 2D variant of the Lare3D [3] application. Both are Lagrangian-remap codes for solving the non-linear MHD equations. The original code development was motivated by the study of solar corona, and their accurate simulation.

A.3 Summary

In this chapter we have outlined the machines and applications used throughout this thesis. We use these applications to demonstrate the capabilities of our developed tools and methodologies.

The use of a selection of supercomputers enables us to test our implementation and analyse application behaviour at large scale. As many memory artefacts are only exhibited at large scale, it is important to capture them in real world scenarios.