Should We Worry About Memory Loss?

O. Perks, S.D. Hammond, S.J. Pennycook and S.A. Jarvis
Department of Computer Science
University of Warwick, UK
{ofjp, sdh, sjp, saj}@dcs.warwick.ac.uk

ABSTRACT

In recent years the High Performance Computing (HPC) industry has benefited from the development of higher density multi-core processors. With recent chips capable of executing up to 32 tasks in parallel, this rate of growth also shows no sign of slowing. Alongside the development of denser micro-processors has been the considerably more modest rate of improvement in random access memory (RAM) capacities. The effect has been that the available memory-per-core has reduced and current projections suggest that this is still set to reduce further.

In this paper we present three studies into the use and measurement of memory in parallel applications; our aim is to capture, understand and, if possible, reduce the memory-per-core needed by complete, multi-component applications. First, we present benchmarked memory usage and runtimes of a six scientific benchmarks, which represent algorithms that are common to a host of production-grade codes. Memory usage of each benchmark is measured and reported for a variety of compiler toolkits, and we show >30% variation in memory high-water mark requirements between compilers. Second, we utilise this benchmark data combined with runtime data, to simulate, via the Maui scheduler simulator, the effect on a multi-science workflow if memory-per-core is reduced from 1.5GB-per-core to only 256MB. Finally, we present initial results from a new memory profiling tool currently in development at the University of Warwick. This tool is applied to a finite-element benchmark and is able to map high-water-mark memory allocations to individual program functions. This demonstrates a lightweight and accurate method of identifying potential memory problems, a technique we expect to become commonplace as memory capacities decrease.

Keywords

Memory, Multi-core, Tracing, Workflow, Simulation

1. INTRODUCTION

In recent years the High Performance Computing (HPC) industry has benefited from the development of higher density multi-core processors. Where only a decade ago multiple processors would have been required to execute programs concurrently, today a single processor chip is able to execute 32 or more processes in parallel. Alongside this substantial growth in compute resource has been a much more modest rate of improvement in memory speeds and DRAM-chip capacities. The effect has therefore been not only a growing disparity between processor performance and memory, but also a sharp decrease in the amount of memory available to each individual processor core. Although this disparity has been well documented, in particular following Wulf and McKee's coining of the term "memory-wall" paper [16], research investigating the impact of declining memory capacity per processing element is still very rare.

An investigation into the runtime changes resulting from a reduced amount of memory per core may also have further benefits than just being able to quantify any increase in execution time. System memory represents a significant upfront cost in the procurement of high-performance computing systems. Depending on the type, capacity and quality of memory required for a system, prices may vary between \$20 and \$75 per GB. When installed in every node of a typical HPC cluster the cost can be significant. Techniques which enable the memory requirement to be reduced may therefore have direct influence on the level of capital expenditure.

Another concern is the power required by memory chips to remain active. Despite recent initiatives to reduce the power consumption of memory subsystems - DDR3 chips provide a reduction in power consumption of 30% compared with equivalent speed and capactiy DDR2 [8] - memory is still responsible for a large proportion of system power. The total power consumption needed for one TFLOP of processing power currently lies at between 320 and 1600 Watts [6].

Such evidence provides a compelling case as to why HPC practitioners should be concerned with memory. In this paper we present a systematic course of action with which to investigate memory use and the impact on runtime of changing memory configurations. Our aim is to be able to measure, understand and experiment with the memory requirements of mixed-science workloads, with a particular interest in the trade-off between increased or reduced memory capacity and its impact on scientific delivery; we would like, if possible, to reduce memory costs, but in order to do this there is a need to understand the associated cost.

We begin the paper by documenting a series of compiler-memory studies, employing compiler toolkits from PGI, GNU, Intel and Sun (now Oracle) and the `memP` profiling tool - a lightweight memory usage analysis tool specifically targeted for the heap profiling of parallel (MPI) applications [2]. We analyse the difference in reported memory high water mark (HWM) between codes built with different compilers; in some cases the HWM exhibited by differently compiled code is potentially significant.

Using memory-usage data obtained from executions of a variety of scientific benchmarks and applications on a large cluster, we present a study which investigates the impact of a reduction in available memory-per-core. In order to cope with a reduction in per-core-memory a distributed code must be run over more processor cores in order to obtain the same amount of memory. Whilst this is possible for codes with a good problem decomposition, some program components such as lookup tables and MPI overheads are fixed, or worse, increase with more processor cores. By artificially fixing the maximum amount of memory-per-core, and combining this with runtime data for different core counts, we are able to assess the impact on runtime for a mixed-science workflow when per-core memory capacity is reduced. In order to ensure that this replicates the behaviour of genuine clusters we employ the Maui scheduler simulator [1] to generate completion times and workflow information.

In the final part of this paper we analyse a single scientific code using new memory tracing tools being developed at the University of Warwick. Our tools allow the memory usage of individual functions to be tracked throughout execution, giving an insight into how the HWM of this specific code is generated. Analysis such as this is likely to become commonplace as memory capacities become smaller and memory accesses become proportionally more expensive when compared to processing costs.

The specific contributions of this work are:

- We present a comparison of the memory requirements of six established scientific benchmarks and applications when compiled with common high performance toolkits. This analysis provides a potential first step in lowering the memory requirements of a code by switching to an alternative compiler toolkit. In our experience this is rare in many HPC sites as users have long-held preferences for particular compiler frameworks and therefore very few have evaluated the performance or memory usage of alternatives. Studies such as this are of significant use in identifying the likely benefits associated with new tools and educating users in these benefits;
- We utilise workflow simulations of benchmarked memory usage and runtime data from a variety of scientific codes to examine the likely impact on workflow completion time when the available memory-per-core is artificially limited. Given that the HPC industry expects to see significant growth in the number of processor cores, but only modest improvement in memory capacity, this study provides insight into the potential effect on HPC workflows in the future;
- We employ a new memory profiling toolkit currently under development to analyse the allocation of memory inside a finite-element benchmark. Our tool is able to record the breakdown of memory to program functions and to map these temporally over the course of execution. If HWM memory requirements are to become problematic, as a number of HPC experts believe, analysis of this kind will become commonplace.

The remainder of this paper is organized as follows: Section 2 describes related work, including general issues of power consumption, programming for memory optimization and a discussion of memory profiling tools; in Section 3 we present a compiler study in which the HWM for a number of applications is investigated under different compiler builds; Section 4 presents a series of studies using the Maui scheduler simulator that show the impact on scientific delivery of a reduction in memory per core; Section 5 presents the memory tracing of a finite element benchmark using a new memory profiling tool currently in development at the University of Warwick. Section 6 concludes the paper and documents further work.

2. RELATED WORK

In most programs, 20-40% of the instructions reference memory [7]. Memory has therefore been subject to much scrutiny over the past 30-40 years. Wulf and McKee's classic work on the "Memory Wall" [16] describes the problems associated with dynamic random access memory (DRAM): the rate of improvement in microprocessor speed exceeds the rate of improvement of DRAM memory speed; resulting in a point at which system performance is totally determined by memory speeds (the so-called Memory Wall). Since the mid 1990s, when this landmark paper was written, several attempts at innovation have been made. These include coupling memory with the processor [13], aggressive on-chip cache hierarchies [9] (such as that seen in the IBM BlueGene series) and 3-D DRAM [12], where memory stacks are situated in close proximity to the microprocessor in order to reduce wire delay between the two.

Generations of architectural innovation have led to lower memory latencies and increased memory bandwidth. Despite this, the power consumption of synchronous DRAM continues to be of concern, particularly in large server- and HPC-environments. Recent DARPA commissioned studies [11] on the challenges for ExaFLOP computing report that the power needed for memory grows linearly with the number of chips (but that the power for the interconnect stays constant) and that memory power grows in relation to the peak FLOP/s per chip. There is clearly benefit therefore in adopting smaller amounts of higher speed memory, and finding ways in which we can trade computation for storage [14]. Previous approaches have been documented using external memory algorithms [15]; such approaches are likely to increase with the general adoption of GPGPUs. Content-aware memory management has also been shown to deliver up to 22% reductions in memory consumption without code modification [5].

Several supporting tools exist to assess memory use. The Performance API (PAPI) allows hardware counters on most modern microprocessors to be accessed; this in turn allows dynamic memory usage information to be collected. Low-level architectural measurements are then correlated with the source/object code through events. A number of high-level tools have been built on top of PAPI, including the HPCToolkit, OpenSpeedShop, PerfSuite, Scalasca, Vampir and TAU. An alternative to PAPI is Valgrind, an instrumentation framework on which dynamic performance analysis tools can be built. The current Valgrind distribution contains a number of production tools, including a memory error detector, a heap profiler and a cache and branch prediction profiler. Valgrind also contains a second heap profiler that examines how heap blocks are used.

In this work we use the tool `memP` [2], a lightweight memory analysis tool specifically targeted at heap profiling MPI applications. We use `memP` because it can be dynamically loaded with very little runtime overhead, and as such it scales well. `memP` provides both heap and stack usage for

	From	Description
POP	LANL	Ocean modelling
miniFE	SNL	Unstructured finite element
Sweep3D	LANL	3D particle transport wavefront
phdMesh	SNL	Unstructured mesh contact search
MG	NASA	Algebraic multigrid solver
LAMMPS	SNL	Classic molecular dynamics

Table 1: Application Benchmarks

	GNU	Intel	PGI	Sun	Var.
POP	33.62	33.67	35.32	-	5.07%
miniFE	180.35	180.34	237.21	180.52	31.53%
Sweep3D	158.34	158.38	158.37	158.34	0.03%
phdMesh	73.64	73.64	91.05	-	23.64%
MG	254.76	255.13	255.01	255.45	0.27%
LAMMPS	58.73	59.23	60.08	58.69	2.37%

Table 2: Memory HWM (MB) for application benchmarks using four different compilers

each process in the execution and returns aggregated statistics for later analysis.

3. MEMORY AND COMPILER CHOICE

In typical HPC environments the choice of a compiler toolkit is dictated largely by those purchased for the system (since not all toolkits provide the backends necessary for code generation) historical preference and, commonly, code runtime performance. We however approach compiler choice from an alternative angle - that of memory usage. At the outset this might seem an odd choice, in that the memory requirements of codes are largely dictated by the programmer's use of program data objects, however, some variation can be identified. Our experience with other full science applications studied with our industry partners is that benchmarked HWM figures vary considerably when more complex codes and problem data sets are assessed. In one recent study of an application a variation of over 225% was reported in HWM between using one compiler and another.

Table 1 presents six mixed-science benchmarks and applications, which cover algorithms and activities common to large computing installations. Memory and runtime benchmarking of each code is performed on an IBM cluster housed at the Centre for Scientific Computing (CSC) at the University of Warwick. The cluster comprises 240 dual-Intel Xeon 5160 dual-core nodes each sharing 8 GB of memory (giving 1.92 TB in total). Nodes are connected via a QLogic InfiniPath 4X, SDR (raw 10 Gb/s, 8 Gb/s data) QLE7140 host channel adapters (HCAs) connected to a single 288-port Voltair ISR 9288 switch, with a core-to-HCA ratio of 4:1. Each compute node runs the SUSE Linux Enterprise Server 10 operating system and has access to the IBM GPFS parallel file system. For our study we use the Oracle Solaris Studio 12.2 tool suite (formerly Sun Studio), the Intel C/Fortran 11.1 compiler suite, GNU 4.1.2 and PGI 10.6, in conjunction with OpenMPI 1.4.2 and the PBS Pro scheduler.

Table 2 shows the effect of compiler choice on memory consumption (HWM) for the benchmarks in question. All benchmarks were compiled using the `-O3` optimisation flag to replicate the behaviour of many HPC users. In each case seen in Table 2, the application benchmarks are run at a fixed core count; 16-cores executed as a 4-node, 4-cores-per-node run. In some cases the difference in HWM is small, 0.03% for Sweep3D. In other cases, phdMesh and miniFE, for example, the difference is significant - 23.64% and 31.53%

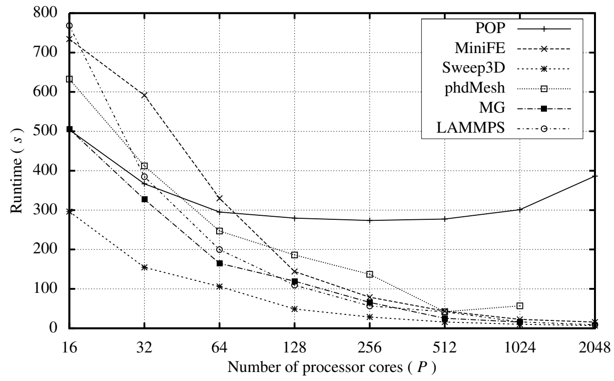


Figure 1: Runtime for the six applications on LLNL Hera

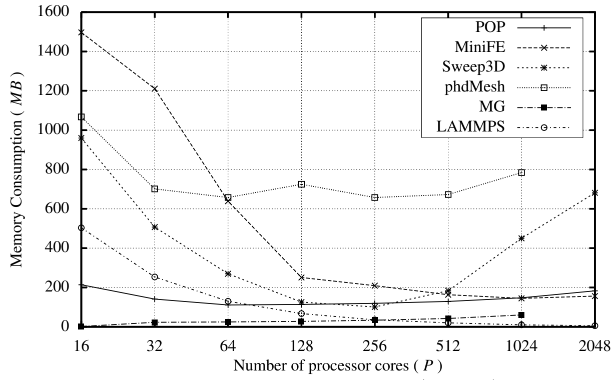


Figure 2: Memory high water mark (HWM) for the six applications on LLNL Hera

respectively. No one compiler consistently outperforms the others in terms of memory HWM. There is also no discernible trend that might inform compiler choice in general.

While this might seem inconclusive, it does suggest one conclusion - application developers must investigate compiler choice.

4. MEMORY AND WORKFLOW

The growth in the number of processing cores per processor chip and slower growth in memory capacities makes it likely that future clusters will exhibit similar architecture layouts to today but will do so with higher core counts and lower amounts of memory per processor-core. In this environment, the scaling of memory will be achieved through the execution of codes at larger core counts, yielding potentially slower runtime but enabling the code to actually be executed. The effects of higher-core count jobs may be lower machine efficiency and the potential lengthening of workflow execution time. With this in mind we are interested in whether it is: (a) possible to continue to run scientific workflows with lower amounts of memory and (b) what the impact of this is on the completion time of a multi-application workflow.

In order to provide a more realistic base on which to conduct experiments we make two assumptions: First, that the workload is mixed science, motivating the use of all six benchmarking codes seen in the previous section, and second, we utilise the Maui scheduler simulation to ensure that the resulting job schedule is representative of how a genuine cluster would execute its workflow.

Experimental data to prime each workflow simulation is taken from executions on the commodity AMD/InfiniBand system called Hera, located at the Lawrence Livermore National Laboratory. Hera uses densely packed nodes consist-

Application	WF-1	WF-2	WF-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%
Total	100%	100%	100%

Table 3: The three workflows used in this study (Workflows 1 to 3) and their composition of jobs

ing of high-performance multi-core CPUs – four-way AMD 2.3GHz Opteron quad-core CPUs (16 cores per node), 32 GB memory per node (2 GB per core). An InfiniBand DDR high-speed interconnect is used for communication between nodes and exemplifies a typical large capacity HPC resource (127 TFlop/s peak).

The runtime and memory requirements for our six applications on Hera can be found in Figures 1 and 2 respectively. As found earlier in this paper, the memory scalability of these codes (as they strong scale) varies considerably. In most cases, the memory consumption decreases as the core count (for an application) increases; it is also true that in most cases the parallel efficiency of the application decreases as the number of cores increases. Thus high core-count jobs decrease run time and parallel efficiency and, when this is coupled with other jobs in the system, the number of available runtime configurations decreases; this will typically increase the overall runtime of a workflow. It is also the case that requests for a higher number of cores typically spend a longer time queueing; this effect we are able to expose directly using the Maui scheduler.

To assess the impact on scientific delivery we combine, through simulation, the benchmarked data into three example workflows (Table 3) in which the mix of jobs is varied. Note that `phdMesh` is excluded from Workflow 3 (WF-3) as this permits analysis of memory configurations below 650MB per core. 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 1.5GB, as this corresponds to the maximum usage seen in Figure 2, and then reduce this to 1280 MB, 1024 MB, 768 MB, 682 MB, 512 MB and 256 MB per core. Many of these cases may seem unlikely, but the choice of 682MB is representative of a hex-core processor with 4GB of memory, for example.

4.1 Using the Maui Scheduler Simulator

To ensure that we replicate as realistic a test environment as possible we use the Maui [1] scheduler (Version 3.3.0) 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. We use 2048 cores of Hera as a basis for our simulated machine (defined using a Maui resource trace file), and our simulated workflows consist of 1000 jobs in the proportions defined in Table 3 (defined by a Maui workload trace file). We maintain the existing polling system in Maui, 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 extra 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. This imposes two constraints

Flag	Value
RPOLLINTERVAL	00:00:30
BACKFILLPOLICY	BESTFIT
RESERVATIONPOLICY	CURRENTHIGHEST
NODEALLOCATIONPOLICY	MINRESOURCE
JOBNODEMATCHPOLICY	EXACTNODE
SIMINITIALQUEUEDEPTH	16
SIMJOBSUBMISSIONPOLICY	CONSTANTJOBDEPTH

Table 4: Maui simulator configuration variables

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

Table 5: Percentage Runtime Increase When Moving from 1536MB to 682MB Per-Core

on our experiments: (i) that the jobs must be submitted in a specified order – we therefore repeat the experiments a number of times with different job orderings to reduce experimental bias; (ii) jobs must be allocated a specific core count when staged – we discuss this further in the next section. Additional Maui configuration values are found in Table 4.

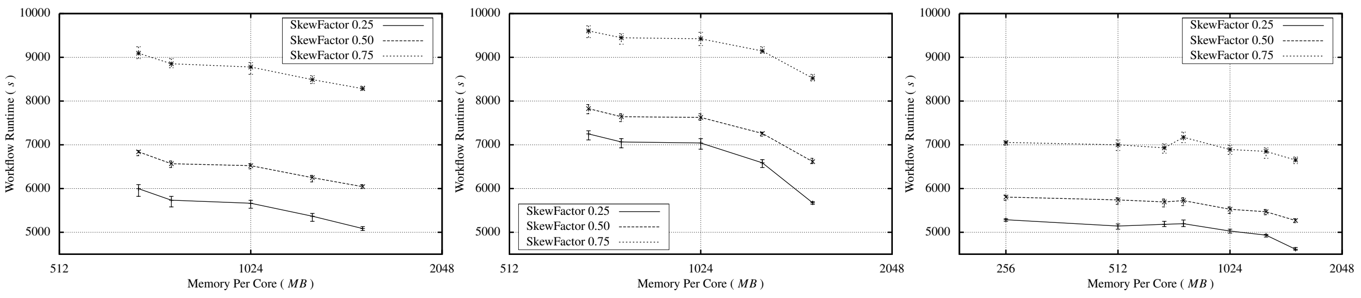
Even when applications are memory constrained they can still be run on various core counts. Typical users will often not select core counts for their jobs optimally. We therefore base the selection of application core count in our workflows on a partitioned Gaussian distribution. The distribution determines 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, a skew of 0.75 will ensure that there is a more even distribution between possible configurations. As we see from the results, the trends are similar no matter what skew factor is used, but this does allow us to consider how users behave (using more cores or fewer cores, usage policies restricting job size, charging models *etc.*) and therefore what effect this may have on the interpretation of the results.

4.2 Results

We compare the time-to-completion for three different mixed-science workflows 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. Each experiment is averaged over ten runs (with randomly generated job ordering in each case) to reduce the effect of any one specific job ordering.

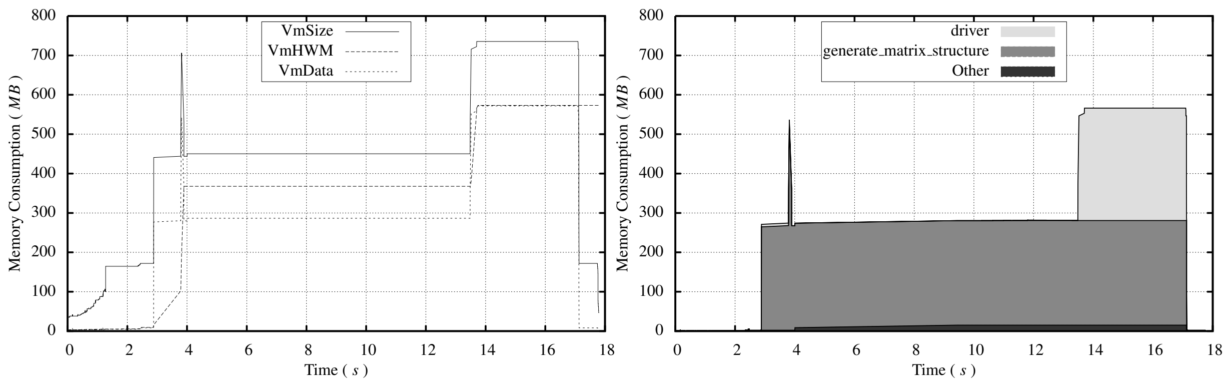
The results for the three different workflows can be found in Figures 3a, 3b and 3c respectively. For each, the average workflow completion time is plotted as a trend line with maximum and minimum times reported as range-bars. Several observations can be made regarding these results:

- In general, decreasing the available memory per core increases the overall runtime. However, this is not a monotonically increasing function, see Figure 3c, for example - in this case the changing core-counts of jobs submitted can create significant decreases in machine efficiency;
- The impact of skew (reflecting usage profile), highlights that the percentage increase in runtime will be more marked the less constrained users are in their choice of core counts. That is, the more users are used to requesting fewer cores, the less absolute difference they will experience in runtime when memory/core is constrained;



(a) Workflow 1 **(b)** Workflow 2 **(c)** Workflow 3

Figure 3: Impact on Simulated Runtime of Reduced Memory-per-Core



(a) System Breakdown of Memory for miniFE **(b)** Memory Trace of miniFE

Figure 4: Analysis of Memory for the miniFE Benchmark

- The rate of increase (in runtime as the memory per core drops) is not constant. This suggests that there may exist opportunities for decreasing memory per core with little impact on scientific delivery, if the workflow mix and system usage profiles allow. Table 5 shows that the impact of moving from 1536 MB/core to 682 MB/core may be a little as 4.19%;

- Workflow mix makes a difference. Table 5 shows that the impact of moving from 1536 MB per core to 682 MB per core (skew 0.25) can be between 12.36% and 27.76% depending on the workflow mix;

- Table 5 also demonstrates that the user's usage patterns are also important. In workflow 2 the percentage increase in runtime can vary by as much as 15.09% depending on how users select core counts.

5. MEMORY USAGE ANALYSIS

The use of analyses such as those described in Sections 3 and 4 highlight that (i) compiler choice may impact the HWM for individual applications, and (ii) that it is possible to calculate in the context of scientific applications the impact of reducing memory-per-core. In this section we present a new tool which is allowing us to determine *why* an application has such a memory HWM. Our aim is that by understanding and capturing memory usage in this way we can improve the future memory requirements of our applications.

Typically, low-level tools such as PAPI and Valgrind might be used to perform cache analysis or to identify memory-performance related problems (*e.g.* cache thrashing or TLB-page misses) with a tool such as memP being used at a much higher level. In this section we analyse the memory behaviour of the miniFE finite element benchmark using a prototype tool currently being developed at the University of Warwick. The purpose of this analysis is to identify which functions within the code contribute the most to memory usage without requiring significant instrumentation or complex tool configuration, as is the case with many system debuggers or profilers. We note that memP claims to have some support for this level of analysis, but our experience has been that this is difficult to obtain. Our tool intercepts each change in memory state through the catching of memory allocation and deallocation calls, recording the operating system reported memory state as well as the program stack. Post-execution analysis of the trace allows us to track each individual allocation and deallocation within the program, attributing the memory requested to each program function. Thus we are able to assess memory allocations at a finer level of granularity (than the whole program) without creating significant overhead in the runtime of the application. Since we also only activate our tracer on changes in the memory state, the tool creates very little overhead during compute-intensive regions of code execution.

Figures 4a and 4b present a breakdown of miniFE's memory from both system and function perspectives. In terms of the system view, the memory requested by the application for data (VmData) shows good correlation with the function breakdown in Figure 4b. Note that we only show the two most significant contributors to memory usage in Figure 4b for reasons of brevity - our tool is able to track memory requests from a much higher number of calls within miniFE, the OpenMPI library and several runtime library functions.

Our memory trace of miniFE (Figure 4b) demonstrates that for the main part of the benchmark less than 300MB of memory are required with peaks of nearly 600MB shortly after the start and in the driver block at towards the end. In our experience many applications demonstrate memory spikes and peaks during specific functions in which data is copied and manipulated, raising the prospect of being able to optimise their memory usage through the generation and processing of data inline (as opposed to explicit allocation, copying and processing). We are currently engaged in further work investigating the potential use of our tool is diagnosing such memory optimisation opportunities within several industry codes.

6. CONCLUSIONS

The declining amounts of memory-per-core arising from the increasing density of multi-core processors and more modest improve in memory capacities has the potential to radically change the way in which HPC applications may need to be constructed. Currently programmers may enjoy as much as 4GB or more of memory per processing core, but future systems may reduce this to 1GB or less. Combined with the high up-front cost of memory during procurement and the additional running costs due to powering large amounts of memory, there is a compelling case for analysing and improving current memory usage.

In this paper we present three separate studies into the use of memory for distributed MPI codes. Our first study investigates the impact that compiler choice can make on memory usage. We demonstrate memory HWM differences of up to 32% for simple benchmarks. In some industry codes we have studied, compiler choice can increase memory HWM by as much as 225%. Where the memory required can be made available within machine resources, users will no doubt prioritise the fastest compiler toolkits. Therefore, we demonstrate that a memory study is also an important aspect of understanding code performance - all to often this is overlooked by users who assume that memory use is identical across all compilers. As memory resources become more constrained this will no doubt need to change.

In our second study we use benchmarked runtime performance and HWM memory data to assess the impact on scientific delivery of a decrease in memory-per-core. We show that there is a relationship between decreased memory per core and an increase in runtime but that, in some cases at least, this is not necessarily a monotonically increasing function. The rate of increase is also not constant and the composition of the workflow as well as usage profile will have an impact on the results. The issues we expose here help to demonstrate that with appropriate data (scheduler logs, scaling graphs for individual applications and HWM data) it is possible to calculate the likely impact on a reduction in memory per core for large supercomputing resources.

The final study in this paper utilises a new memory profiling and tracing tool currently in development at the University of Warwick. This tool is able to intercept memory-state changing events such as allocations and deallocations to record memory usage at a per-function level. We have demonstrated excellent levels of correlation with the HWM and VmData statistics obtained from the Linux kernel giving confidence in our tools' ability to provide high levels of code coverage. The purpose of presenting this tool is to show that where individual applications show potential problems (perhaps using techniques such as those seen in studies 1 or 2), tools are becoming available that can help map these problems to specific entities in the application itself. Although declining memory-per-core will no doubt cause difficulties for some applications, we are becoming well placed to identify these difficulties in advance and to change the development of applications before such problems become significant.

Our appeal to Programme Managers is simple: Application scientists may demand 4 GB/core. Measure current usage with tools such as `memP`; investigate different compiler options if possible. Then investigate the potential impact of reducing memory; clearly this will have some overhead, but the trade-off between more CPU hours and lower total cost of ownership may be a price worth paying.

Acknowledgements

We are grateful to Scott Futral, Jan Nunes and the Livermore Computing Team for access to, and help in using, the Hera machine located at the Lawrence Livermore National Laboratory. We also acknowledge support of the Centre for Scientic Computing at the University of Warwick and the UK Science Research Investment Fund.

7. REFERENCES

[1] `Maui Cluster Scheduler`, 2010. `http://www.clusterresources.com/products/maui-cluster-scheduler.php`.

[2] `memP`, 2010. `http://sourceforge.net/projects/memp/`.

[3] S. Biswas et al. PSMalloc: Content Based Memory Management for MPI Application. In *Proceedings of the 10th MEDEA workshop on MEmory performance: DEaling with Applications, systems and architecture*, pages 43–48, New York, NY, USA, 2009. ACM.

[4] D. Dunning, R. Mooney, P. Stolt, and B. Casper. Tera-Scale Memory Challenges and Solutions. *Intel Technology Journal*, 13(4):80–101, 2009.

[5] J. Hennessy and D. Patterson. *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, 2003.

[6] Hewlett-Packard. *DDR3 Memory Technology*, 4 2010.

[7] S. Iyer et al. Embedded DRAM: Technology Platform for the Blue Gene/L Chip. *IBM J. Res. Dev.*, 49:333–350, March 2005.

[8] P. Kogge et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Technical report, DARPA, September 2008.

[9] G. H. Loh. 3D-Stacked Memory Architectures for Multi-Core Processors. *SIGARCH Comput. Archit. News*, 36:453–464, June 2008.

[10] D. Patterson et al. A Case for Intelligent RAM. *Micro, IEEE*, 17(2):34 –44, 1997.

[11] H. D. Simon. The Greening of HPC - Will Power Consumption Become the Limiting Factor for Future Growth in HPC? HPC User Forum, October 2008.

[12] J. S. Vitter. External Memory Algorithms and Data Structures: Dealing with Massive Data. *ACM Comput. Surv.*, 33(2):209–271, 2001.

[13] W. A. Wulf and S. A. Mckee. Hitting the Memory Wall: Implications of the Obvious. *Computer Architecture News*, 23(1):20–24, 1995.