D2.4.1 – Alternative use of fat nodes

WP2: Underpinning and cross-cutting technologies

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exascale computing, software and simulation

Due date:	M24
Submission date:	30/09/2013
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	UEDIN
Version:	1.0
Status	Final
Author(s):	Alistair Hart (CRAY), Harvey Richardson (CRAY), Dan Holmes (UEDIN), Pekka Manninen (CRAY), Michèle Weiland (UEDIN)
Reviewer(s)	Andreas Gerndt (DLR) and Mats Aspnas (ABO)

Dissemination level	
PU	*PU - Public*

Copyright © CRESTA Consortium Partners 2013

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	16/08/2013	Prepared deliverable outline	Michèle Weiland (UEDIN)
0.2	27/08/2013	Adding contributions on co-location, microkernels and asynchronous progress engine	Michèle Weiland (UEDIN)
0.3	28/08/2013	Adding Offload Servers Section, plus Exec Summary, Intro and Conclusions	Michèle Weiland (UEDIN)
0.4	29/08/2013	Integrated updates by Harvey & Dan	Michèle Weiland (UEDIN)
0.5	30/08/2013	Final draft for internal review	Michèle Weiland (UEDIN)
1.0	25/09/2013	Final version for submission after internal review	Michèle Weiland (UEDIN)

Copyright © CRESTA Consortium Partners 2013

Table of Contents

1 EXECUTIVE SUMMARY 1

2 INTRODUCTION 2

2.1 PURPOSE 2
2.2 GLOSSARY OF ACRONYMS 2

3 BACKGROUND AND MOTIVATION 3

4 CO-LOCATION OF HPC WORKLOADS 5

4.1 INTRODUCTION 5
4.2 METHODS 6
4.2.1 Benchmark codes 6
4.2.2 Hardware used 6
4.2.3 Running the single applications 7
4.2.4 Application characterisation 7
4.2.5 Co-location 8
4.2.6 Co-location metrics 10
4.3 RESULTS 10
4.3.1 Effect of varying node speeds 12
4.3.2 Comparing CPU architectures 13
4.4 CONCLUSIONS AND FURTHER WORK 13

5 OFFLOAD SERVERS 16

5.1 SERVER LOCATION 16
5.2 SOFTWARE IMPLEMENTATION OPTIONS 17
5.2.1 Standard API approach 17
5.2.2 Generic Offload approach 17
5.3 OFFLOAD SERVER DEMONSTRATOR IMPLEMENTATION 17
5.3.1 The offload server API 18
5.3.2 Implementation details 19
5.3.3 An example 20
5.4 OTHER USES OF OFFLOAD SERVERS 21
5.5 CONCLUSIONS 21

6 BACKGROUND PROCESSING OF MPI COMMUNICATION 23

6.1 BEHIND THE SCENES: MPI MESSAGE PROTOCOLS AND OVERLAPPING COMPUTATION AND COMMUNICATION 23
6.2 PERFORMANCE OF THE ASYNCHRONOUS PROGRESS ENGINE 25

7 MICRO-KERNELS 27

7.1 MOTIVATION 27
7.1.1 Core-specialisation with Monolithic Kernel OS 27
7.1.2 Asynchronous Progress Engine with Non-Blocking MPI Communication 27
7.1.3 Motivating Scenario for Separate OS Module for Disk I/O 28
7.1.4 Motivating Scenario for Locality-Aware Memory Management OS Module 29
7.2 METHODS 29
7.2.1 Customisation of a Standard Monolithic Kernel 29
7.2.2 Micro-kernel Operating Systems 29
7.3 CONCLUSIONS 31

8 FUTURE WORK 32

9 CONCLUSIONS 34

10 REFERENCES 35

Copyright © CRESTA Consortium Partners 2013

Index of Figures

Figure 1: Kernel benchmark characterisation, running on 2.1 GHz processors with 1600 MT/s memory speed. 9

Figure 2: Application client/server distinction 18

Figure 3: The non-blocking communication feature of MPI allows in principle overlapping communication with other work. 23

Figure 4: The eager messaging protocol potentially allows overlapping. The case of blocking point-to-point communication is shown on the left, the non-blocking case is shown on the right. 24

Figure 5: For larger messages the overlap is not available. The case of blocking point-to-point communication is shown on the left, the non-blocking case is shown on the right. 24

Figure 6: The Cray Asynchronous Progress Engine uses specific communication threads for more complete overlap. 25

Figure 7: Overlap availabilities of non-blocking point-to-point and collective communication. 26

Figure 8: Schematic of operating system components in a typical monolithic kernel and microkernel [8] 30

Figure 9: Comparison of Compartmentalisation in a Monolithic Kernel, a Microkernel and a Hybrid Kernel [9] 31

Index of Tables

Table 1: List of Cray XE6 node hardware configurations considered. 6

Table 2. Co-location metrics M_1 for the CLASS=C problem running on increasing number of nodes with Hardware Configuration 2. Results are colour-coded, with green showing good co-location and red denoting poor co-location. 11

Copyright © CRESTA Consortium Partners 2013

1 Executive Summary

This report summarises the work that was undertaken in Task 2.4 “Alternative use of fat nodes” as part of CRESTA’s WP2 on “Underpinning and cross-cutting technologies”. More specifically, the report presents research into different ideas for the use of fat nodes on future systems, ranging from practical to more speculative approaches:

- Co-location of workloads;
- Offload servers;
- Background processing of MPI communication;
- Micro-kernels.

After an introduction that provides further detail on the purpose of both the Task and the report, the following points are addressed in turn:

- A background Section motivates the research into alternative uses for fat nodes and provides information on the evolution of hardware that has led to current system architectures, where core counts per node increase and the memory per core decreases.
- Following this, the report looks in detail at the co-location of HPC workloads on tightly coupled HPC systems, evaluating both the performance impact and the practical issues related to multiple applications sharing the same hardware resources. The performance tests were done using the NAS Parallel Benchmarks, which consist of kernels that are representative of a CFD application.
- The report then outlines the idea of using spare cores on a fat node as offload servers with dedicated tasks, thus freeing up compute cores. The example that is given in this report is that of IO servers. We also describe an RPC-like API to explicitly offload tasks.
- The following chapter describes how spare cores can also be used to actively progress asynchronous communication of MPI applications, thus overlapping communication and computation, focussing in particular on Cray’s latest MPI implementation. This Section also introduces a metric to quantify the amount of overlap that can be achieved.
- The report then moves on to describe the idea of micro-kernels. Unlike monolithic operating systems, micro-kernels consist of separate task entities, which can be scheduled independently and thus use spare cores for OS operations.
- The final two Sections describe possible future work continuing on from the research presented in this deliverable, as well as some conclusions that we can draw. The future work is largely concerned with how the approaches presented here could be applied to the CRESTA co-design vehicles as well as to power management issues.

© CRESTA Consortium Partners 2011

2 Introduction

The purpose of this deliverable is to summarise the research that was undertaken as part of Task 2.4 "Alternative use of fat nodes" inside WP2 of CRESTA. Fat nodes are already a reality in modern HPC systems, and the reduced amount of memory per core on these nodes means that some applications are already at the stage where they cannot exploit all the cores on a node. Under-populating nodes is a common technique used to increase memory per process (and more efficiently use available memory bandwidth) and can be optimal for performance.

This deliverable investigates how we can use spare cores on a fat node in a useful and productive manner. Four different avenues are explored and described in subsequent Sections. The ideas outlined in this document (which vary in maturity and feasibility) will be shared and discussed with the owners of the CRESTA co-design vehicles.

2.1 Purpose

The purpose of this public deliverable is to gather and present the results from Task 2.4 of WP2 of the CRESTA project.

2.2 Glossary of Acronyms

API	Application Programming Interface
CFD	Computational Fluid Dynamics
CG	Conjugate Gradient
CPU	Central Processing Unit
CRESTA	Collaborative Research Into Exascale Systemware, Tools and Applications
D	Deliverable
DRAM	Dynamic Random Access Memory
ECC	Error Correcting Code
EP	Embarrassingly Parallel
FT	Fourier Transform
GPU	Graphics Processing Unit
GROMACS	Groningen Machine for Chemical Simulations
HPC	High-Performance Computing
IFS	Integrated Forecast System
I/O	Input/Output
MG	Multi-grid
MPI	Message-Passing Interface
NUMA	Non-Uniform Memory Architecture
OS	Operating System
PCI	Peripheral Component Interconnect
PE	Processing Element
RPC	Remote Procedure Call
TLB	Translation Lookaside Buffer
WP	Work Package

3 Background and motivation

Over the last decade, the number of floating-point units ("cores") per physical CPU die has increased dramatically. This has been driven by the so-called "power wall": previously, increasing the clock speed delivered improved computational performance per CPU. However this also raised the power consumption and introduced a new engineering challenge of cooling the CPU so that the heat generated could be dissipated. Eventually, this meant that designers looked for alternatives to making further significant increases in the clock speed. Additional performance gains instead came from building CPUs with more, lower-frequency floating-point cores. Recent improvements in CPU design have improved their power efficiency so that clock speeds have once more shown a modest increase, but this will not offset the trend towards multiple cores. All of this has contributed to the continued trend of Moore's Law, with transistor counts doubling every two years, and House's variant of overall CPU performance doubling every 18 months. Modern (as of 2013) multi-core CPUs have 8 or more cores per CPU, whilst many-core GPU accelerators and coprocessors extend this trend even further.

This rapid increase in the number of computational cores has not, however, been matched by a commensurate increases in the shared node resources used to support the potential floating-point calculation rate, viz. total on-node memory size (typically 32 or 64GB per node); memory bandwidth used to pull data from the main memory into the fast, on-die caches; and the network bandwidth. Memory bandwidth is particularly important for many applications: like CPU floating-point performance, DRAM speeds have also shown exponential growth over the last 15-20 years, but with a doubling time of 3 years rather than 18 months [1] [2]. As a consequence, memory bandwidth has become an increasingly significant limiter of application performance, and this trend is likely to continue. Network performance will show some improvements over the next decade, but the gains will not be exponential. Large message transfer rates are limited by the bandwidth of the bus (e.g. PCI Express) connecting CPU and NIC, which will not improve dramatically. Smaller messages (arguably more important as we strongly scale[1] fixed-size problems to run on Exascale machines) are more sensitive to the latency of the interconnect, and this has long remained around 1μs and is unlikely to decrease further.

Given these limitations, it is clear that for many codes we can achieve the same overall performance per node with fewer "active" cores per node running a portion of the calculation. We will generically refer to such a portion as a Processing Element (PE), the precise meaning of which depends on the programming model used. A PE could be an MPI rank, a Fortran coarray image, or an OpenMP thread associated with a rank or image or similar. With current CPUs, some codes see similar (or even improved) performance when running on 50-75% of the available cores per CPU. As hardware trends continue, this proportion is likely to decrease, as limited shared resources limit the accessible floating-point performance. Currently, inactive cores automatically go into a sleep state, consuming less power. Future Exascale-era CPUs may take this further, allowing the hardware, OS, runtime or application to completely power down these cores (so-called "dark silicon"). With power constraints paramount in an Exascale-class supercomputer, this is an attractive option.

A relevant, but complicating, factor here is the increasingly deep hierarchy of boost and sleep states available in modern CPUs. The hardware can detect idle cores and reduce their clock speed (and thus power consumption). When sufficient cores are in these sleep states (and the environment permits), the active cores can then have their clock speeds temporarily boosted within the same overall CPU power envelope. With current

---

[1] By strong scaling, we mean dividing a fixed-size global problem over increasing numbers of PEs, with the local problem size per PE decreasing accordingly. This is in contrast to weak scaling, where the local problem-size remains fixed.

© CRESTA Consortium Partners 2011

CPUs, this can amount to as much as 500MHz of additional performance for the active cores. For this reason, we can actually surpass the "all cores active" performance per node with a "half-dark" version. It is not obvious to what extent this boost factor will feature in the Exascale era, where the aim of idle cores is to limit the overall power consumption, rather than to enable boosting of other cores.

Boosting aside, switching cores off will undoubtedly save power, but it is not necessarily the most productive thing to do. Supercomputing systems have base-line power consumption regardless of the CPU activity, and even sending half the cores "dark" will not halve the power consumption of the system. It is therefore reasonable to consider whether idle cores could be productively used, rather than simply powered down.

It is important here to clarify what we mean by "productive use". We take the view here that an Exascale supercomputer will focus on computational problems that demand such a capability resource. We focus, therefore, on three specific scenarios: using the spare cores to improve the performance of the application running on these nodes at the OS and runtime level; improving single-application performance through PE placement; and improving ensemble-based simulation throughput through co-location. This deliverable discusses different potential scenarios for the productive use of fat nodes, looking at options which are feasible today as well as more speculative future developments.

© CRESTA Consortium Partners 2011

4 Co-location of HPC workloads

Application co-location is the simultaneous running of two or more HPC applications on a set of nodes, such that each application gets a partial share of the resources available to each node. The goal of such an approach is to improve the combined performance of the applications, by reducing the overall runtime and/or energy consumption. Architectural trends suggest this may become an increasingly promising method for improving application performance as we approach Exascale computing. In this Section, we explore application co-location for tightly coupled HPC supercomputers. By considering the runtime performance when pairs of codes are co-located on a variety of node architectures, we show that co-location can be beneficial and indicate in which real-world situations it can improve performance.

4.1 Introduction

Co-location of two or more independent simulations on a set of nodes may improve overall throughput. In addition, many Exascale-potential applications can be regarded as coupled simulations of two or more interacting physical systems, each using a distinct set of PEs. Co-locating PEs from these different simulations on a node can also improve application performance.

The argument for why this may be a sensible thing to do is simple: if different sets of PEs on the node are doing different things and accessing different shared node resources, then each resource is less likely to be saturated and less likely to hinder overall performance. For coupled simulations, it is very likely to be true that different sets of PEs are behaving independently and not competing for the same resources at the same time. For ensembles, it is possible that the two sets of PEs are actually doing the same thing. But, with no barriers between the independent sets, they can spontaneously get sufficiently "out-of-step" that the same situation occurs.

Of course, there are complications. Most real algorithms use all the shared resources to some degree, so seemingly independent computational tasks are not actually that different in terms of resource usage, so co-location can be less successful than expected. Likewise, even if the tasks do make independent usage of shared resources, the success of co-location will depend on how much time is spent on each task and the frequency and pattern of the tasks in the application timeline.

Nonetheless, it is reasonable to investigate whether there are any circumstances in which application co-location can be successful on current tightly-coupled HPC systems and, if so, to see whether any guidelines can be given to indicate when co-location might be a productive thing to try for a particular code. Application co-location has been investigated in the past [3][4]; in this report we consider a number of new angles on this issue.

First, most studies concentrate on generic cluster architectures, rather than the tightly coupled architectures that are more typical of likely Exascale architectures. This point should not be underestimated: systems with a tightly coupled design are generally designed to allow only one application at a time to use node resources and without interference. As a result this can make it difficult to investigate co-located workloads. Here we concentrate on the Cray XE6 supercomputers. In particular, the faster interconnect used in these systems changes the balance of network communication time compared to computation.

The range of systems used is shown in Table 1. The systems used are company-owned systems used to investigate customer issues, and consequently contain a wide variety of CPUs. We also take advantage of this heterogeneity to compare performance on a wide variety of processor architectures and clock speeds, coupled with different memory speeds. It is unusual to have access to such a wide variety in one study. Being able to directly compare this range of node configurations allows us to understand and model the effect of these hardware parameters on code performance.

© CRESTA Consortium Partners 2011

Based on the expected hardware trends, we can therefore extrapolate our findings towards expected Exascale supercomputer designs.

Ref.	CPU				Memory	
	Manufacturer and model	Codename	Part	Clock (GHz)	Size (GB)	Speed (MT/s)
1	AMD Opteron	Interlagos	6272	2.1	32	1333
2	AMD Opteron	Interlagos	6272	2.1	64	1600
3	AMD Opteron	Interlagos	6272	2.1	128	1600
4	AMD Opteron	Interlagos	6274	2.2	32	1333
5	AMD Opteron	Interlagos	6276	2.3	32	1333
6	AMD Opteron	Interlagos	6281	2.5	64	1600
7	AMD Opteron	Abu Dhabi	6380	2.5	32	1333
8	AMD Opteron	Abu Dhabi	6380	2.5	64	1600
9	AMD Opteron	Fangio	6275	2.3	32	1600

Table 1: List of Cray XE6 node hardware configurations considered.

4.2 Methods

It is very difficult to analyse the co-location properties of real applications, which typically feature a number of phases in their execution, with each phase relying on the node resources in different ways. It is therefore better to focus on smaller, kernel benchmarks that typify the individual phases.

4.2.1 Benchmark codes

We use the well-established NAS Parallel Benchmarks v3.3[2], and concentrate on the kernel benchmarks: CG (conjugate gradient), EP (embarrassingly parallel), FT (Fourier transform) and MG (multigrid), which we compile with the latest released version of the Cray Compilation Environment (CCE), version 8.1.

The codes were compiled using the provided Makefile. The build allows us to define a global problem size using the build option `CLASS=<size>`, where `<size>` is encoded by a letter. We considered `CLASS=A,B,C` and `D` for this study, but mainly concentrate on `CLASS=C` for the quoted results. The number of MPI ranks is fixed at compile time using the `NPROCS=<ranks>` build option.

4.2.2 Hardware used

We begin by briefly describing the hardware used for the tests. It is important to do this first, as it informs some of the co-location decisions that will be made following.

All tests were done on Cray XE6 supercomputers located in the Cray Data Center at Chippewa Falls, Minnesota, USA. The Cray XE6 is a tightly coupled supercomputer, coupling compute nodes together via the low-latency, high-bandwidth Cray Gemini interconnect. Each node contains two 16-core AMD Opteron processors, giving 32 cores per node. The two CPUs on each node are generally identical, but there are a range of options from the 6200-series "Interlagos" and 6300-series "Abu Dhabi" processors. Each node shares banks of DDR3 memory (usually totalling 32 or 64GB per node), with the DIMMs supporting a maximum transfer rate of 1333 or 1600 MT/s.

The downside of this variety node designs is that the number of nodes with a given hardware configuration is more limited; in this study we used up to either 16 or 32 nodes, depending on availability.

---

[2] Available from: http://www.nas.nasa.gov/publications/npb.html

© CRESTA Consortium Partners 2011

The AMD Opteron CPUs are based around eight dual-core "modules", called "Bulldozer" for the 6200-series Interlagos Opterons and "Piledriver" for the 6300-series Abu Dhabi CPUs. In each case, a module couples two x86 out-of-order processing cores, which share some structures including higher-level memory caches and, more importantly here, floating-point units (FPU). Each FPU can separately process one 128-bit instruction (e.g. SSE) or they can be unified to process a single 256-bit-wide AVX instruction. This is an important feature, as it means that certain compute-intensive codes can give the same performance per node using one core per module (8 cores per CPU, or 16 cores per node, sometimes known as "single stream mode") as when using both cores in the module (16 cores per CPU, or 32 cores per node and "dual stream mode"). In certain circumstances, single stream mode can even lead to higher performance as, when half the cores on a CPU are idle, the hardware may take advantage of this to temporarily boost the clock speed of the active cores (whilst remaining within the CPU's designed power envelope).

It is very difficult to predict *a priori* whether a given code, even if computationally-intensive, will give better performance per node in single stream mode. The answer may even depend on how the code is run.

4.2.3 Running the single applications

The benchmarks were executed on the compute nodes using the Cray ALPS library command aprun. Two modes of running were considered. Firstly, each code was compiled using K MPI ranks (i.e. K PEs, as the codes are pure MPI), and then run in dual stream mode, using all the floating point cores on the node (with flag "`-N32 -j2`" for `aprun`). Each code internally reports correctness and relevant runtime, which it converts to a performance figure. It is this runtime that we consider in the results in this report. Secondly, we recompile with K/2 ranks and then run again using single stream mode (flags "`-N16 -j1`" for `aprun`).

On AMD CPUs on the Cray XE6, the CG, EP and FT benchmarks performed best in dual stream mode. MG is more complicated, as single stream mode is faster in some, but not all cases. For instance, for `CLASS=C`, we find single stream mode to be faster when running with 1 node (using 16 ranks), 4 nodes (64 ranks) or 8 nodes (128 ranks), but dual stream gives better performance with 2 nodes (also 64 ranks) or 16 nodes (512 ranks). The precise reasons for this oscillating behaviour are hard to establish, but it does have bearing on the co-location results that follow. It is interesting to note that where single stream mode is better, the advantage over dual stream mode is generally only a few per cent, but where single stream mode is worse, it is typically 20% slower. This suggests that we should perhaps look at this as single stream mode being especially inefficient in these cases, rather than the other way round.

4.2.4 Application characterisation

To understand why co-location of two kernels might be advantageous, we need to characterise how each application makes use of the shared resources of the node by attempting to model the application runtime in terms of time spent on: floating-point computation, waiting for data from memory and network transfers. To do this, we profiled the codes using the Cray Performance Analysis Tool (CrayPAT), selecting to trace all user routines and MPI (options "`-u -gmpi`" for the `pat_build` command). The CrayPAT API was used to restrict profiling to the timed region of the benchmarks. We then run the instrumented code and generate the profile. The network time is reported under "`MPI`" in the main profile, while the "`USER`" time consists of time spent in the different computational routines. The "`ETC`" heading reports time spent in lower-level operations, which was consistently extremely small and can be ignored. None of the benchmarks does any significant I/O, so we do not need to consider this here.

The next step is to separate the USER time into two parts; that constrained by floating-point calculation speed; and that constrained by memory bandwidth. If the CPU clock-speed is C MHz, and the DDR3 memory DIMMs have a transfer rate of M MT/s (mega Transfers per second), we can most simply model the user time U for a given code at a given node count by:

$$U = \frac{A}{C} + \frac{B}{M}$$

where A and B are constants. To find A and B most simply, we took advantage of the diverse range of Cray XE6 nodes available in the Cray Data Center. We ran (and profiled) the benchmarks on five different Cray XE6 variants, all with AMD Interlagos CPUs, as listed in entries 1 to 5 of Table 1.

We then performed a least squares minimisation to estimate A and B from these 5 values of U. To estimate the error on A and B, we did the following crude analysis. Given the best-fit A and B, we can predict a value of U for each of the 5 architectures, and calculate the percentage error based on the actual values of U. We then take the maximum percentage deviation across the 5 data points as the overall error on the fit, and ascribe this same fractional uncertainty to A and B.

Clearly this is an overly simplistic model. We are assuming that any hardware-controlled boosting of the CPU clock speed is done in proportion to the base clock speed and that all 5 hardware combinations boost in exactly the same amounts. Similarly memory-caching effects are only accommodated if we make similar assumptions regarding cache bandwidth. Nonetheless, we find this model works surprisingly well, with the percentage errors at the level of a few per cent, and we can then use it to estimate the fractions of U that are clock- and memory-bound.

In Figure 1 we show the application characterisations for each benchmark as we strong-scale a particular problem size (`NPB CLASS=C`) across increasing numbers of cores. In many ways the results are as expected given an understanding of the applications: EP is extremely clock-bound; FT makes greatest use of the network; as we strong-scale to higher numbers of PEs, the network becomes more important. What is more surprising is that the characterisations of the benchmark kernels are more similar that we might expect, despite the very different computational tasks carried out. This is significant; the success of co-location relies on subsets of PEs on each node stressing different shared resources at any given times. It is already clear here that co-locating seemingly different computational algorithms is no simple guarantee of this and any success will depend on when resources are accessed in each PE's timeline.

4.2.5 Co-location

We are studying co-location to understand how shared resources on the node can be more-smoothly utilised by a mixed workload. Given the compute module architecture of the AMD CPUs, we therefore opt to co-locate pairs of applications on a node such that they each take one core of each module. We also tried an alternative approach, where we located PEs for the two applications on different CPUs on the node. As the results were extremely similar, we do not discuss this case explicitly in this report. For Intel CPUs, we take a similar approach, co-locating applications so each uses one of the two hyper-threads running on a physical core.

Co-locating codes on the cores of a tightly coupled supercomputer such as the Cray XE6 or Cray XC30 is not easy. Users typically demand the optimal and reproducible performance that comes from having exclusive access to a compute node and the system software reflects this. The Cray ALPS library does permit the running of two or more binaries together in MPMD mode (Multi-Program, Multi Data) sharing, for instance, a common MPI_COMM_WORLD communicator, but only if all the cores on a given node host PEs from the same binary. To investigate co-location, we need to go beyond this and mix PEs from different applications on the same nodes. There is (at least currently) no simple way to do this beyond editing the two applications to become part of a single binary. For the NPB kernel benchmarks, we did this as follows.

© CRESTA Consortium Partners 2011

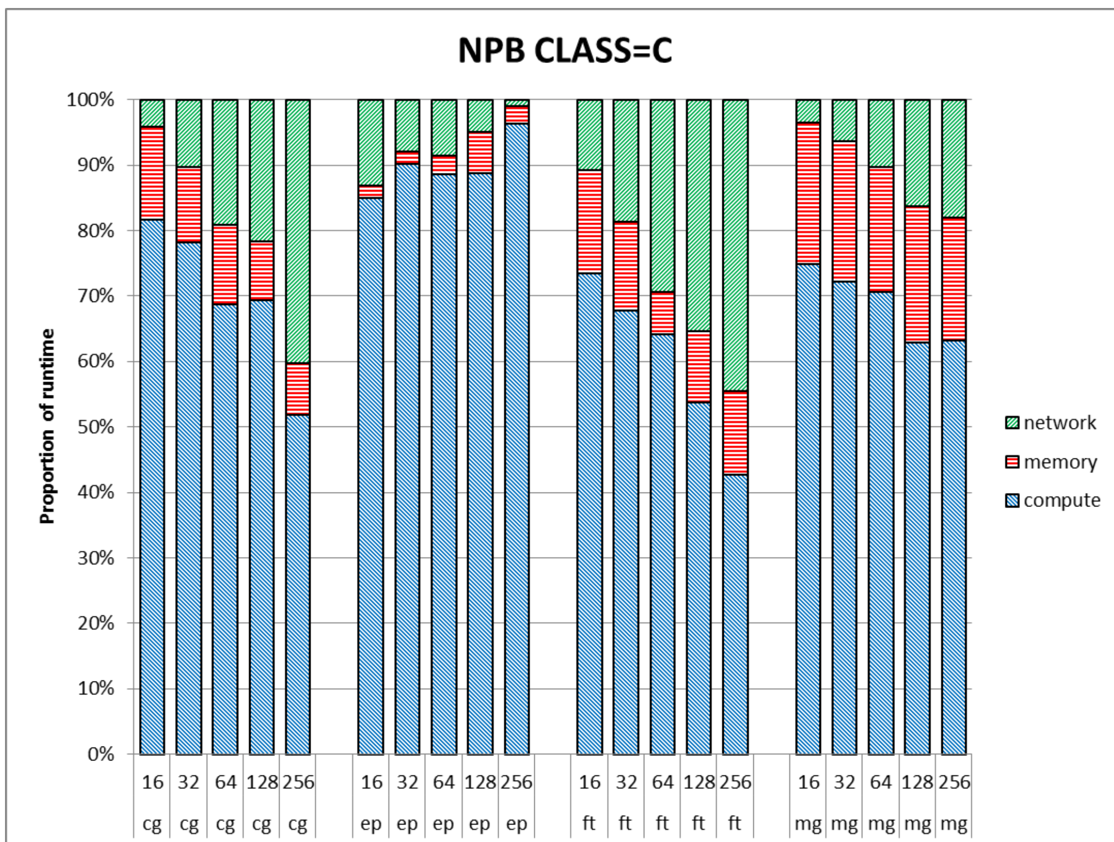

Figure 1: Kernel benchmark characterisation, running on 2.1 GHz processors with 1600 MT/s memory speed.

Two MPI applications can be combined into one by using a common main program that initialises the MPI library, and then splits the default `MPI_COMM_WORLD` communicator and runs each application in one of the sub-communicators, before calling `MPI_Finalize()` when all called applications finish. The applications themselves are modified by changing their main program into a subroutine which takes an argument that is the MPI sub-communicator handle to be used by that application (and stored in an appropriate shared data structure. Calls to `MPI_Init()` and `MPI_Finalize()` in the application are removed, and references to `MPI_COMM_WORLD` are replaced by the appropriate sub-communicator handle. This modification to an application is easily tested using a separate, three-line main program that calls: `MPI_Init()`; followed by the subroutine that is the previous main program, with argument `MPI_COMM_WORLD`; and then `MPI_Finalize()`. The code should then execute exactly as before.

The complication comes if there is a symbol clash in the two sets of object files (usually because both contain subprograms with the same name). Possible solutions here are to rename subprograms in the source code, modify the object files to remove clashes or to somehow restrict the objects in each application to separate namespaces. For the Fortran codes we considered, namespacing was most easily done by textually inlining (via `INCLUDE` statements) the existing source files into an application-specific MODULE, defined in a separate file and then in the main program exposing only the top-level subprograms with "`USE..ONLY`" statements.

This modification was made for all the kernels, allowing any combination of two applications to be run. An additional change was that the top-level subroutine returns as an argument the time taken to run the benchmark portion of the application. This allows the driver program to (optionally) repeatedly run the benchmark until a specified time has been spent (plus benchmark initialisation time for each instance). This is helpful when co-locating applications with different execution times. It is also important here because we are not interested in the co-location performance of the NPB kernels *per se*, but rather because we hold them to be representative of different parallel computational tasks in a more-realistic application (recall that they are representative of

components of a CFD application). As such tasks are likely to be repeated many times, this repeated-run mode of kernel execution gives a better picture of the success of co-location in the real case.

Clearly this manual fusing of application codebases is feasible for the limited kernel benchmark suite we consider here. For more realistic applications, a more user-transparent method would almost certainly be needed if co-location is to be widely exploited.

The combined binary is then executed on the compute node using the aprun command in single-stream mode (`"-N32 -j2"`), ensuring that the rank placement is such that each compute module hosts one rank from each of the original applications, and each node runs 16 ranks from each code. Effectively we are overlapping the two codes, each running in single stream mode, as a single dual stream job.

Having run each code multiple times are part of this co-location job, we average the benchmark runtimes as reported by the code. We found these runtimes to be very consistent, so we did not need to worry about outlying results biasing the mean.

4.2.6 Co-location metrics

Having run binary combinations of the benchmarks, we need to define a metric to assess the success of the co-location approach. In the general case, this metric would probably be an appropriate function of both the application runtimes and the energy consumed in running these codes. As we are not considering energy usage in this report, the metric is based solely on application runtime as follows.

Assume that we co-locate applications X and Y. Running solo and taking full advantage of the node (in the better of single or dual stream modes) they take times S_X and S_Y respectively. When co-located and each using half the cores on each node, they take times T_X and T_Y. There are two obvious ways to measure success. If, as we do here, we envisage these benchmarks as being part of larger applications, we will be co-locating the benchmarks multiple times. In this case, the best metric for co-location success is:

$$M_1 = \frac{S_X}{T_X} + \frac{S_Y}{T_Y}.$$

Alternatively, if we view our benchmarks as complete applications that are only going to be run once, then we need to compare the times for running the applications separately in sequence, to the time of the slower of the pair when co-located:

$$M_2 = \frac{S_X + S_Y}{\max(T_X, T_Y)}.$$

In either case, values greater than unity indicate a net benefit from co-location. As justified previously, we will confine our discussion to metric M_1 in what follows.

4.3 Results

In Table 2 we show the M_1 co-location metrics for pairs of NPB kernel benchmarks calculating the CLASS=C-sized problem on increasing numbers of Cray XE6 nodes on Hardware Configuration 1 (see Table 1). The cells of the Table are shaded such that the well-co-locating combinations are coloured green and poorly-co-locating pairs are red. For each node-count, we display the result of co-locating a given pair of benchmarks in a (necessarily symmetric) co-location metric matrix.

What is immediately clear is that there are many cases where co-location is successful at the level of 10-15%, and occasionally more. The cases where it is successful also outnumber those where co-location is detrimental. This in itself is encouraging. If no other clear predictions are possible, the results do suggest that it is worth trying co-location if at all possible.

There is also a general trend that poor co-location tends to disappear as we strongly scale to more cores. This is as expected; the local problem sizes are smaller in these

cases and the computation therefore places less stress on the various shared node resources, leading to less opportunities for contention. The network becomes increasingly important *relatively* as we strong scale to smaller local problem sizes, but it is the *absolute* amount of data transferred that leads to contention, and this is decreasing.

1 node	CG	EP	FT	MG
CG	1.07	1.11	1.05	0.98
EP	1.11	0.88	1.11	1.22
FT	1.05	1.11	1.09	1.08
MG	0.98	1.22	1.08	0.89

2 nodes	CG	EP	FT	MG
CG	1.17	1.12	1.14	0.95
EP	1.12	0.90	0.98	0.80
FT	1.14	0.98	1.10	0.91
MG	0.95	0.80	0.91	0.73

4 nodes	CG	EP	FT	MG
CG	1.12	1.11	1.18	1.08
EP	1.11	1.01	1.13	1.02
FT	1.18	1.13	1.25	1.14
MG	1.08	1.02	1.14	1.04

8 nodes	CG	EP	FT	MG
CG	1.24	1.17	1.43	1.16
EP	1.17	1.01	1.31	1.04
FT	1.43	1.31	1.61	1.34
MG	1.16	1.04	1.34	1.07

16 nodes	CG	EP	FT	MG
CG	1.18	1.11	1.27	0.99
EP	1.11	1.02	1.29	0.99
FT	1.27	1.29	1.52	1.08
MG	0.99	0.99	1.08	0.98

Table 2. Co-location metrics M_1 for the CLASS=C problem running on increasing number of nodes with Hardware Configuration 2. Results are colour-coded, with green showing good co-location and red denoting poor co-location.

The worst co-location performance tends to be when codes are paired with MG. We saw previously that MG performed better in single stream mode than when using all cores on the CPU. This suggests it is particularly sensitive to node resource contention.

© CRESTA Consortium Partners 2011

Given MG's characterisation above, this is likely to be due to sharing the memory bandwidth. This sensitivity is probably preventing co-location from being successful.

Apart from MG, benchmarks CG and FT tend to co-locate well with themselves and other benchmarks. EP can show a minor negative effect when co-locating. This may well be a consequence of EP being very clock-bound, whilst CG and EP have more dependence on the memory speed and network.

The main conclusion we can draw from these results is that there is often a modest co-location advantage, except where we are co-locating very clock-bound, floating-point intensive codes or codes that have particular problems running in single stream mode. The overall advantage may not (currently) be enough to justify merging distinct applications to increase workload throughput. For coupled simulations, however, it is almost certainly worth exploring whether PEs from different subsets should be co-located on the node. This can usually be done without modifying the code; a runtime file (typically `MPICH_RANK_REORDER`) can be used to change the placement of the PEs on the cores of the various nodes without recompiling the code. A 5-10% improvement in application performance is not insignificant and (if parts of the code are network-bound) should improve application scalability.

4.3.1 Effect of varying node speeds

Given the range of Cray XE6 node configurations available in this study, it is interesting to investigate how the co-location metrics change as we vary either the CPU clock speed or the memory bandwidth. Whilst the range over which these can be varied is modest and we are fixed with the AMD Interlagos CPU architecture, understanding the trends in the values of the co-location metric may give some indication as to whether any advantage is likely to persist as CPU architectures evolve towards the Exascale.

We first look at the effects of varying the CPU clock speed, comparing the co-location metric results for the CLASS=C-sized problems running on Hardware Configurations 2 and 6. This changes the clock speed from 2.1 GHz to 2.5 GHz, whilst fixing the memory speed at 1600 MT/s.

Surprisingly, the co-location metric results are nearly identical for all the co-location pairs and node counts considered. EP shows a slight increase in co-location advantage when paired with itself, but only at certain node counts. Other than that the co-location metrics agree to within a couple of percent.

The conclusion here is that clock speed does not strongly influence co-location success if we are primarily interested in runtime. It may, of course, be more interesting if we instead measure energy consumption as part of our co-location metric.

Varying memory speed does have a bigger effect. If we compare Hardware Configurations 1 and 2, which both have clock speeds of 2.1 GHz but change the memory speed from 1333 MT/s to 1600 MT/s.

We find a clear pattern emerge here. At small node counts (1 or 2 nodes) we see a strong improvement when co-locating any code with EP, with co-location metrics improving as we increase memory speed, by 5-10% for heterogeneous pairs and as much as 30% when EP is co-located by itself. Other metric values remain essentially unchanged. This may seem surprising, as EP is the least memory bound of the codes, but this allows the other code in the co-location pair to take full advantage of the increased memory bandwidth. This effect decreases at higher node counts when less memory bandwidth is needed.

More striking at higher node counts (8 or 16 nodes) is that the co-location metric for any co-location pair including FT shows a decrease of around 20% when we increase the memory speed. The current trend is for memory speed to increase more slowly than the floating-point performance of a CPU, so it is likely that memory bandwidth will become a more serious limitation as we move towards the Exascale. The results in this Section reinforce that there is a good argument for using co-location, especially where

an application has compute-heavy parts of the code that allow the memory bandwidth to be used almost exclusively by other, co-located tasks.

4.3.2 Comparing CPU architectures

Thus far, we have focused on Hardware Configurations using Interlagos CPUs. It is interesting to see what effect changing the CPU type has on co-location. We consider two different AMD CPU versions. The first is the next-generation Abu Dhabi processor. This uses the “Piledriver” compute module that offers some incremental improvements on the Bulldozer module used in the Interlagos CPU, which should improve application performance, especially for clock-bound codes.

The other is a special purpose variant of the Interlagos CPU known as "Fangio". Each standard Bulldozer module can process 8 double precision floating-point operations per clock cycle. In the Fangio processor, this is capped to 2 double precision operations per clock. The peak performance of the CPU is thus reduced by a factor of four, but with the same memory bandwidth etc. Floating-point-bound codes will be sensitive to this loss of peak performance, but more memory-bound codes (including many CFD applications) may perform as well on Fangio as on Interlagos [5]. As reduced floating-point performance should translate to lower power consumption, Fangio is a very useful prototype for future CPU architectures used in Exascale systems.

The exact node configurations are shown as Hardware Configurations 7 to 9.

The co-location experiments were carried out in the same way as before. As with Interlagos, we find that varying the clock speed of the Abu Dhabi CPUs does not noticeable affect the co-location metrics.

When comparing co-location metrics on Abu Dhabi to Interlagos (Hardware Configurations 8 and 6), the differences are small, with variations at around the 5% level and with few clear patterns. Application pairs involving CG perform marginally better on Abu Dhabi CPUs at small node counts. Co-locating EP with itself also degrades the co-location advantage, possibly because EP can best exploit the improved floating point performance in the newer CPU, which then increases the code's reliance on unchanged factors, notably memory bandwidth.

Only small numbers of Fangio nodes were available in the Cray XE6 systems, but on these the only significant change is for co-location pairs including EP. Running on Hardware configurations 9 and 5, we find that when EP is paired with CG, itself or FT we see a reduction in the co-location metric of 5-10%. It is hard to draw definitive, partly because of the modest node counts available and also because Hardware Configurations differ not only in CPU architecture but also in memory speed.

In general, given the relatively minor architecture differences between AMD Interlagos, Abu Dhabi and Fangio CPUs, it is hard to identify underlying trends that indicate what may happen at the Exascale.

4.4 Conclusions and further work

As reported in the previous Subsections, we have measured how well the NAS Parallel kernel benchmarks perform when run in tandem on nodes of a tightly coupled supercomputer, the Cray XE6 with various node configurations. The measure of success considered here is based purely on execution time.

Taking advantage of the wide range of node architectures available within a single HPC system, we examined the effects of CPU clock speed and memory bandwidth on the co-location and also how changing the CPU architecture altered the co-location properties. We used a simple performance model to characterise the codes, dividing their runtime into clock-bound computation, memory bandwidth bound execution and network-bound time.

The main conclusion is that co-location was successful, increasing runtime throughput for a wide range of pairs of kernels by around 10%. This effect was most striking when the local problem size was relatively large (i.e. small node counts in our strong scaling

studies). This is as expected, when the greatest strain is being placed on the shared node resources.

It is difficult to extract general principles to predict co-location success from this study. This is partly because the interaction of the code with the hardware is complicated, but also because a real-world application will contain a number of different tasks stressing different components of the hardware. Whilst a given task might be characterised by one of the kernels used in this study, the co-location success will depend on how long is spent in each task and the sequence and frequency in which the various tasks are called.

It is clear, however, that if a code prefers to run in single stream mode (one core per module on AMD CPUs, or one hyper-thread per code on Intel CPUs), it is unlikely to co-locate well.

We saw that the clock speed has a relatively minor effect on the co-location metric, but memory speed is more important. As the rate of increase of CPU performance is outstripping that of memory bandwidth, this suggests that co-location will indeed be more important as we approach the Exascale. This is compounded by fact that the increase of CPU performance is largely coming from increased core count rather than inflated performance per core. With more cores there is greater scope for co-location.

We studied three (admittedly similar) AMD CPU architectures in an attempt to further identify hardware trends affecting co-location success as we progress towards the Exascale. The architectural similarity, however, made it hard to see definitive, long-term trends that could be extended this far into the future. The comparisons did, however, underline that there are opportunities for co-location success on all the CPU architectures considered.

What was clear from this study was that the complications of launching two codes in a co-located fashion are currently considerable on many tightly coupled HPC systems, and simpler mechanisms would be needed if this procedure were to be more widely adopted.

We considered co-location to CPU resources, but coprocessors and GPUs are widely regarded as being important in this area; not only do they increase floating point performance on the node, but they also (potentially) do this with reduced energy consumption compared to a CPU. Such accelerators are increasingly being added to supercomputer node architectures such as the Cray XK7, which replaces one of the AMD CPUs per node with an Nvidia Kepler K20X GPU. The execution model of the GPU requires the main program (at least) to reside on the CPU, with computational kernels being offloaded to the GPU. Typical GPU codes only use one core of the CPU for this, with the remainder sitting idle. There is clearly a good case here for co-location, running a CPU code on the 15 idle cores.

As before, the measure of success here could be based either on execution time or energy consumed. It would be interesting to compare these approaches, as hardware becomes available.

We considered the co-location potential of the CRESTA co-design vehicles. GROMACS and IFS are good candidates for full-application co-location investigations. The GROMACS algorithm is quite task-based and science calculations typically require an ensemble of similar simulations that could be co-located. IFS couples a number of physics models in a simulation and there may be advantages to co-locating PEs involved in different models on the same node.

Looking forward towards the Exascale, we expect the trend of fat nodes getting fatter makes co-location a more interesting proposition. Comparing co-location results on Interlagos CPUs with those on Abu Dhabi and, especially, Fangio, it appears that the benefits of co-location are relatively robust against architecture changes and we are perhaps therefore justified in believing they will persist as CPU architectures evolve towards those needed for an Exascale supercomputer. What is also clear from this study is that CPU architectures are already very complicated, and that it is difficult to

make sweeping generalisations about which codes might benefit from co-location and by how much. CPUs will undoubtedly get more complicated, if only to satisfy the widely quoted power cap of 20MW for an Exascale supercomputer. We therefore believe that co-location studies such as this should be revisited as processors change, to track whether the benefits suggested here will truly remain in the Exascale timeframe.

© CRESTA Consortium Partners 2011

5 Offload servers

In this Section we will consider the potential for using spare cores on a fat node to run offload servers. This should allow us to use spare compute and network capacity to progress aspects of an application at the same time as the application continues with computational phases. The most obvious practical example of this approach (and one we use as motivation and demonstrator) is that of I/O servers.

An I/O server implementation can offload filesystem activity (which can be time consuming) in a way that brings performance benefits for the following reasons:

1. There is opportunity for overlap of the I/O with application computation either due to faster communication between the application data space and the I/O server than direct application I/O or because of explicit asynchronous communication with the I/O server
2. I/O from a (limited) set of I/O servers to the filesystem may be more efficient than from all application processes.

We need look no further than the CRESTA co-design vehicles to find that I/O is a challenge to scalability. For example:

- **GROMACS**
  GROMACS is the major open source and free software package for biomolecular simulation developed as an international collaboration steered from KTH, Sweden. This application can achieve exceptional efficiency due to a combination of algorithm choice and tuning. However the I/O implementation, which gathers a large amount of data to one node, is becoming a bottleneck.
- **OpenFOAM**
  OpenFOAM is an open-source software package for computational fluid dynamics (CFD). It has a scalability challenge due to its output scheme where many files are written per process. This is not efficient as the number of processes increases and the application makes more demands on the filesystem [6][7].

Two approaches to address these I/O challenges are to use a more coordinated approach (for example MPI-I/O) or to perhaps use an I/O server implementation to limit filesystem activity to a small set of processes (note that MPI-I/O does aggregation already).

Although I/O servers are not a new idea, the increasing trend towards fatter nodes and access to systems with both fat-nodes (24-32 cores) and the capability for fast single-sided communication means that now is the ideal time to revisit this concept.

In the following Subsections we consider how offload servers might be placed and which APIs should or could be supported. We also introduce a new API, which we believe would be a good method for incorporating the use of offload servers into an application in the most general way.

5.1 Server Location

We will consider a multi-process application and choose to define the "client" as an application routine or possibly a progress thread. The offload server is a process that can send and receive data to and from the application clients. The server executes as part of the application (one of the processes forming the application) and, in the case where it is an I/O server, the server has access to the filesystem.

Various location strategies are possible for the offload server:

1. It can be co-located with the application as progress thread(s);
2. It can be a separate process located on nodes only used for offload servers;
3. It can be a separate process co-located on a node with standard application processes.

Here, we will concentrate on the case of separate processes forming offload servers.

5.2 Software Implementation options

Enabling an application to take advantage of offload servers with minimal programming effort would be the ideal scenario. There are two approaches that can fulfil this requirement:

1. An API that performs a standard function (for example I/O to a file) which uses offload servers transparently.
2. An API that allows the application to start and interact explicitly with an offload server implementation. Within this context a more standard API could be provided.

We now consider these cases in more detail.

5.2.1 Standard API approach

An example of how the standard API could work would be through a proxy of a standard API such as POSIX I/O or MPI-I/O. Calls to that API would then operate via the offload servers. There is a major difficulty with this approach in that we need some mechanism to start up the application along with the offload servers in a way that is not visible to the application. For example, with MPI it may be possible to start multiple binaries, with some of these binaries starting in a context with a fake `MPI_COMM_WORLD` and an extra known communicator that includes the offload servers. Such a solution is very system specific, but minimally invasive.

If we can proxy MPI-I/O, it would be worth considering having the I/O server located on the same node as the MPI-I/O aggregators, which may offer an extra performance advantage.

5.2.2 Generic Offload approach

In this generic approach the offload framework is made explicit to the application at the expense of some extra coding in the application. For an MPI application, it is necessary that the application can progress using a new communicator for its computation ranks. We chose this more generic approach, as outlined in the following Section.

5.3 Offload server demonstrator implementation

We have developed a demonstrator implementation for offload servers for MPI applications. The goals of this implementation were as follows:

1. Simplicity of use;
2. Flexibility in operations supported by the offload server;
3. Support for MPI applications;
4. Allow asynchronous progress of offload server operations.

Our approach was to develop a generic API that looks like an RPC API. This is described in more detail below.

We decided to start with a Fortran implementation for the following reasons: firstly to see how easy it was to have a complete Fortran stack; secondly because we wanted to make sure that we could use function pointers (something more familiar to C programmers); and finally because we could use standard Fortran coarray syntax for single-sided data communication and control. Having the capability for single-sided operation is fundamental to supporting asynchronous data movement.

The implementation segments the application processes into two sets as illustrated below:

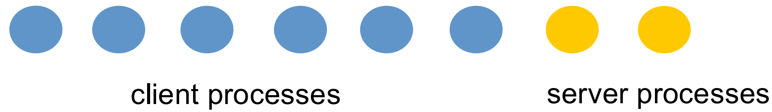

Figure 2: Application client/server distinction

In this example the application is split between 6 compute processes (clients) with two processes reserved for use as offload servers, which have been placed at the highest ranks of the MPI application.

Our implementation works as follows:

The application registers a callback function, which is to be used later when the application wants to make use of the offload servers. It then calls an initialisation routine that allows the application to split into compute processes/ranks and offload servers. From that point on, the application should continue by only using the compute processes and the programmer should keep in mind the distinction as per Figure 2. The application can make a call to the offload server implementation to asynchronously start the callback function. The function will be executed on both the compute node and on an offload server. The function can provide or reference data, which is communicated behind the scenes by the implementation. For example, an array could be provided to be written to a file from an offload server.

The next Sections describe the API and the implementation in more detail.

5.3.1 The offload server API

Applications using the offload server implementation must include the module `os`.

A callback function is registered using the `os_register` subroutine call

Name: OS_REGISTER(FUNC,HANDLE,SERVER_USAGE,DIRECTION)
Description: Register function for later use as callback
Arguments:

FUNC	Function to be registered
HANDLE	Integer handle returned by call
SERVER_ USAGE	Optional. Integer used to define which server can be used to handle requests, must have one of the following values SERVER_SAME (use same server) SERVER_ANY (use any server) n (User virtual server number n – starting with 1)
DIRECTION	Optional. Integer used to define direction of transfer to or from server. Currently ignored (direction is TO the server) but must be supplied in the case that the implementation will move the data.

The handle it returns is used later. Note that this procedure must be called on all processes. The function passed in must match one of the following prototypes:

SUBROUTINE FUNC(key,data,flag)
or SUBROUTINE (key,flag)
Arguments:

KEY	An integer key (or tag)
DATA	Variable of type os_data used to point at user data to be moved
FLAG	logical variable set to false if the executing rank is an offload server

To start up the server infrastructure the `os_start` procedure must be executed and it is defined as follows:

Name: OS_START(COMM,COMM_COMPUTE,NSERVERS,FLAG)
Description: Starts up the server infrastructure
Arguments:

COMM	MPI Communicator of processes that call OS_START
COMM_COMPUTE	(out) MPI Communicator for compute ranks

© CRESTA Consortium Partners 2011

NSERVERS (in/out) Number of servers requested/provided
FLAG Logical flag, value true denotes a server rank

On return, this routine either returns a communicator for the computation (clients) or it returns `MPI_COMM_NULL`. Note that on the server the call enters a request-handling loop so will only return when all work is done. The application should continue and use the returned `comm_compute` where `MPI_COMM_WORLD` may have been used before. If the flag argument is not present then the requested number of servers will be placed at the highest ranks in MPI_COMM_WORLD. If the flag is present then any rank making the call with the flag set to `.true.` will become an offload server.

As the application progresses it can now use the server infrastructure by initiating the function callback using the `os_func_start` call.

Name: OS_FUNC_START(HANDLE,TAG,REQUEST)
Description: Start the function callback and data movement related to handle HANDLE
Arguments:
HANDLE Integer handle of previously registered function
TAG Integer tag that will be passed on function callbacks
REQUEST (out) Integer request number

Calling this subroutine will cause the implementation to call the handler callback function on the client, collect any data to be transferred, move it to the server and then call the same registered function on the server providing it with the data. Note that data is only moved by the implementation behind the scenes if the "data" form of the function was registered. The `os_data` datatype includes various pointers as shown below.

```
type os_data
  integer,          dimension(:), pointer :: i => null()
  real,             dimension(:), pointer :: r => null()
  double precision, dimension(:), pointer :: d => null()
  character*(:),    pointer :: c => null()
  type(os_data),    pointer :: p => null()
end type os_data
```

The provided function should associate these pointers with data that needs to be transferred. Note that the key (tag) can be used to help differentiate multiple calls.

In the case where the programmer wants to move the data then the `get_active_client_rank()` routine may be called *on the server* to return the rank of the active client.

The program can test that the calls have completed (on both ends) with the `os_func_wait` call:

Name: OS_FUNC_WAIT(HANDLE,REQUEST)
Description: Waits until a request has completed
Arguments:
HANDLE Integer handle of previously registered function
REQUEST Integer request matching previously started function

When all work is complete for a client the `os_finalise` call must be called. This call will terminate the offload servers.

5.3.2 Implementation details

We chose to start with a pure Fortran/MPI implementation but wanted to have the ability to overlap the communication and the function execution on the server with computation on the clients. We decided to use Fortran coarray syntax to achieve this.

The function register calls are made both on clients and servers and hence there is global knowledge of the handle-to-function relationships. A data structure is kept on each server with details of client request states. The clients write into this structure when there is work to do and this work is done during the server loop. When a request is completed the most recently completed request number is written back to the client. Both client and server data structures use locks for coordination.

Coarray put and get operations are used to move status and data information. In the case where the `os_data` structure is used, the server can directly request data from the client using coarray syntax.

5.3.2.1 Possible Future Work on the API

Some restrictions of the current implementation are that only one request can be active at any time per server and that there is no collective support (multiple clients mapped to a single function call on the server). The former limitation is easy to lift if we keep state for multiple requests at a time. We will also investigate adding specific domain APIs that can be handled transparently by the infrastructure.

Also, data movement is currently only supported from client to server. Extending this to data movement in both directions is a small change.

Another feature that we have not yet implemented is a combination of a data mover API and advice on use of coarray primitives that can be used directly in client- and server-executed functions to move data without setting or accessing the pointers in the `os_data` structure.

Our implementation is Fortran with MPI. To support applications written in C a version using either MPI single-sided or OpenSHMEM could be created.

5.3.3 An example

The following example shows how the API could be used to copy array data to a server, and have the server write the data to a file.

```
program test
use mpi
use os
implicit none
integer :: rank,size,comm,ier,nservers,handle
integer :: newrank,newsize,i,key,request
integer,target :: int_data(100)
type(os_data) :: data
integer :: server_usage

int_data=[(i,i=1,100)]

call MPI_init(ier)
call MPI_Comm_rank(MPI_COMM_WORLD,rank,ier)
call MPI_Comm_size(MPI_COMM_WORLD,size,ier)

int_data=[rank,0,0,(i,i=4,size(int_data))]

server_usage=SERVER_SAME
call os_register(data_handler,handle,server_usage,1)
nservers=1
call os_start(MPI_COMM_WORLD,comm,nservers)

if (comm != MPI_COMM_NULL) then

 call MPI_Comm_rank(comm,newrank,ier)
 call MPI_Comm_size(comm,newsize,ier)
 print *, "rank",rank," new rank and size",newrank, newsize
 key=100+rank
 call os_func_start(1,key,request)
 call os_func_wait(1,request)
end if

call os_finalise
call MPI_Finalize(ier)
```

```
contains

 subroutine data_handler(tag,data,am_computer)
 integer, intent(in) :: tag
 type(os_data) :: data
 integer, dimension(:), pointer :: int_arr
 logical, intent(in) :: am_computer
 character*30 fname
 integer, save :: call_no
 integer unit

 if (am_computer) then
   data%i => int_data
  else
   int_data = data%i
   ! We have the data from the client

   call_no = call_no + 1
   write(fname,"('A',i4.4)")call_no
   open(file=fname, newunit=unit, form="UNFORMATTED")
   write(unit)int_data
   close(unit)
  end if

 end subroutine data_handler
end program test
```

The program starts by initializing the `int_data` array which will eventually be written to disk via the offload server. It then registers a data handler routine (`data_handler`) using the form with an argument of type `os_data` which will be used to move the data. The program then calls `os_start` to split into compute and server processes. The compute processes then call `os_func_start` to initiate the client and server callbacks. After waiting for this to complete all processes call os_finalise() When `os_func_start` is called the implementation calls the callback function (`data_handler`) on the client, moves the data and then calls the function on the server. This routine both provides the data (by associating the `data%i` pointer) and sends it to a file on the server using the same pointer associated by the implementation.

5.4 Other uses of offload servers

An obvious application for offload servers is to handle I/O. An area that may be of interest would be to implement user checkpointing on top of the existing infrastructure. Furthermore, we could add two new features:

- Checkpointing to multiple servers using ECC across servers in order to be able to recover from failure;
- Checkpointing to memory.

This could be extremely useful if delivered as part of a fault tolerant MPI solution.

5.5 Conclusions

We have developed a new API which we think is the simplest way to experiment with an offload server architecture for an MPI application but is designed to allow asynchronous usage and allow overlap with computation. It is minimally invasive for the application requiring that a callback function be registered, that the application can

continue with a new communicator and that data to be moved can be associated with a provided data structure or communicated directly during the callback.

© CRESTA Consortium Partners 2011

6 Background processing of MPI communication

In this Section we consider how extra cores on a node might be used to improve the performance of asynchronous message-passing communication libraries. In the case where extra cores are available and there is opportunity for communication overlap, there is a good case for using spare cores to progress the communication asynchronously with the application's computation. As we started this work, Cray released a new version of the MPI library that supports an asynchronous progress thread. We have used this implementation to validate our approach using simple examples.

6.1 Behind the scenes: MPI message protocols and overlapping computation and communication

The MPI API provides many functions that allow point-to-point messages (and with MPI 3.0, also collective communication) to be performed asynchronously. Ideally, applications should be able to overlap communication and computation and hide all data transfer behind useful computation. Unfortunately, this is not always possible at the application level, and further, not always possible at the library implementation level.

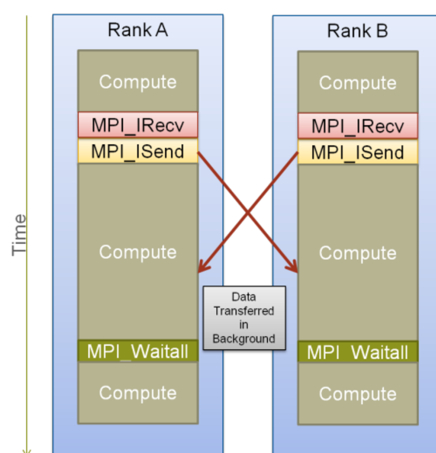

Figure 3: The non-blocking communication feature of MPI allows in principle overlapping communication with other work.

In reality, even though the MPI library has asynchronous API calls, overlap of computation and communication is not always possible. This is usually because the sending process does not know where to put messages on the destination, as this is part of `MPI_Recv`, but not `MPI_Send`.

As an example, let us consider the different interconnects of the Cray XE and XC product series, referred to as Gemini and Aries respectively. On these networks, complex tasks, such as matching message tags with the sender and receiver, are performed by the host CPU. This means that message-matching is always performed by one fast CPU per rank. Therefore, messages can usually only be progressed when the program is inside an MPI function. Some applications insert extra `MPI_Probe` calls to make this happen, however without a separate progress "thread" it is difficult to do this in practice. Furthermore, it is not ideal to have that progress thread share the CPU resources that are being used for computation.

Smaller messages can circumvent this issue when using the "eager" message transfer protocol: if the sender does not know where to put a message, it can be buffered until the receiver is ready to take it. When `MPI_Recv` is called, the library fetches the message data from the remote buffer and into the appropriate location (or potentially local buffer), and the sender can proceed as soon as data has been copied to the buffer. The sender will block if there are no free buffers.

© CRESTA Consortium Partners 2011

In the case of non-blocking communication, data is copied from the buffer into the real receive destination when `MPI_Wait` or `MPI_Waitall` are called. This approach involves an extra memory copy, but features much greater opportunity for overlap of computation and communication.

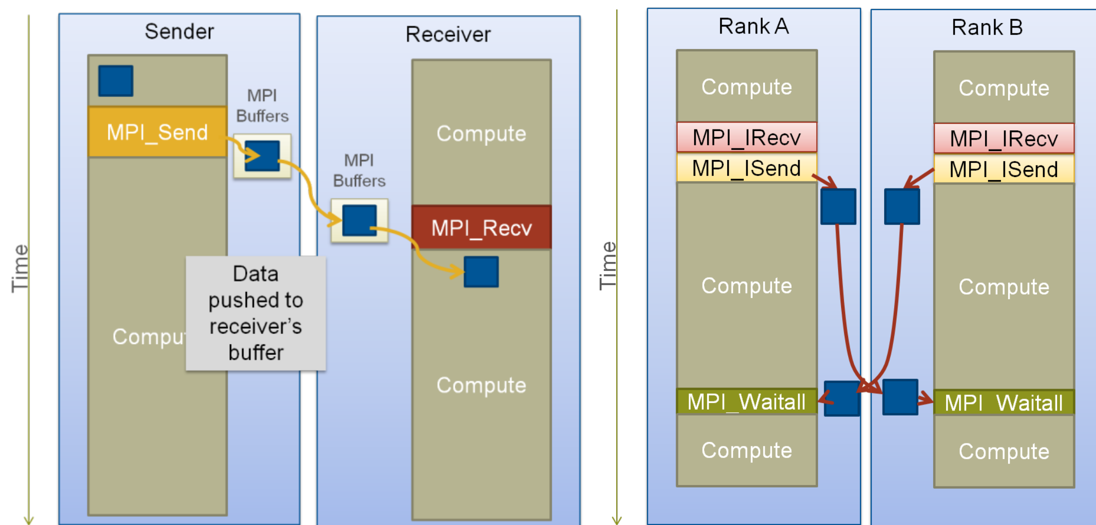

Figure 4: The eager messaging protocol potentially allows overlapping. The case of blocking point-to-point communication is shown on the left, the non-blocking case is shown on the right.

However, for larger amounts of data the communication uses the rendezvous data transfer protocol. With this protocol, no similar temporary buffers are being used and data transfer often only occurs during the `MPI_Wait` or `MPI_Waitall` statement. When the message arrives at the destination, the host CPU is busy doing computation, so is unable to do any message matching. Control only enters the MPI library when `MPI_Waitall` occurs and does not return to the application until all the message data is transferred. In other words, there has been no overlap of computation and communication.

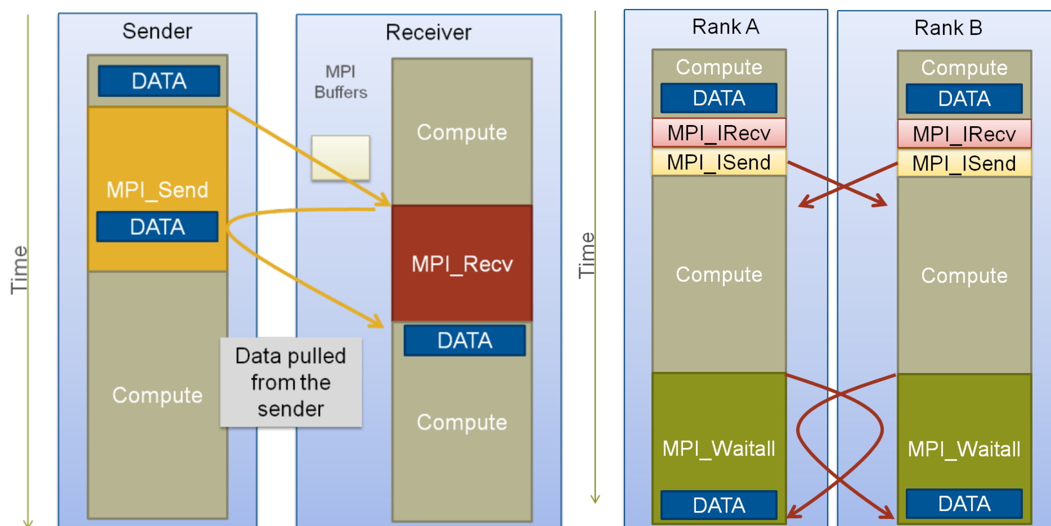

Figure 5: For larger messages the overlap is not available. The case of blocking point-to-point communication is shown on the left, the non-blocking case is shown on the right.

The MPI libraries that progress eager and rendezvous messages differently usually also allow for controlling the threshold for a message to become eager. However, sending more messages via the eager protocol places more demands on buffers on receiver, and if the buffers are full, transfer will wait until space is available or until the `MPI_Wait`.

On the Cray XC and XE systems, there is also an opportunity to spawn additional threads that allow progress of messages while computation occurs in the background. Each MPI rank starts a "helper thread" during `MPI_Init`. These threads may run on

cores especially allocated for this purpose, or they can utilise core oversubscription (e.g. “hyper-threading” on Intel CPUs). These threads progress MPI operations (performing message matching and initiating the transfers) while the application computes. Consequently, data has already arrived by the time Waitall is called, so overlap between compute and communication has occurred. The Cray Asynchronous Progress Engine operates only on inter-node rendezvous messages, since eager messages already have a possibility for overlap. On real-world applications, performance improvements of 10% or more have been reported.

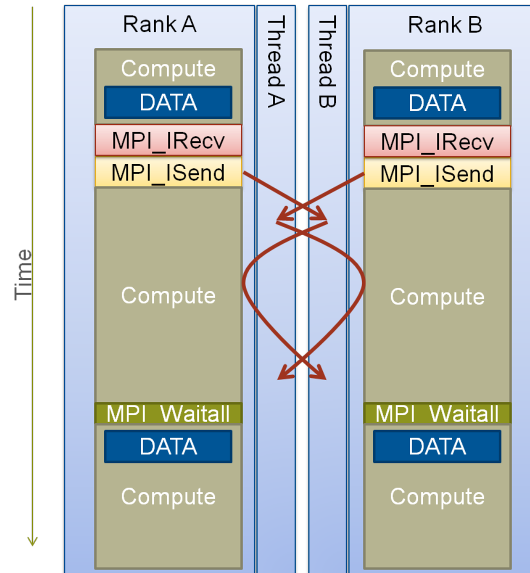

Figure 6: The Cray Asynchronous Progress Engine uses specific communication threads for more complete overlap.

6.2 Performance of the Asynchronous Progress Engine

Let us define an “overlap availability time” *O* as

$$O = (C + N) - T$$

where

- *T*: a time required for completing a communication operation and an overlapped computational task
- *C* (computation time): time required for performing the computational task alone
- *N* (network time): time required for completing the communication operation alone.

Further, by defining

$$\text{ovl}\% = \frac{(C + N) - T}{C + N - \max(C, N)} * 100 = \frac{O}{\min(C, N)} * 100$$

ovl%(overlap availability %) would be zero in case of no overlap (the case with blocking MPI communication) and 100 in case where the communication overhead has completely been hidden (*T* being equal to either the computation or network time, whichever is larger). Negative value would mean performance penalty from trying the overlap.

In Figure 7 (left), the overlap availability of a “message chain” communication pattern, realized with `MPI_Irecv`, `MPI_Isend` and `MPI_Waitall`, with and without using the Asynchronous Progress Engine is presented. The platform is a Cray XC30, and we use 256 MPI tasks (occupying 16 nodes each having 16 Intel Sandy Bridge cores). The computational task is a matrix-multiply of size yielding a computational time roughly similar order of magnitude with the communication time. Without the Asynchronous Progress Engine, overlapping is in fact causing performance degradation.

The non-blocking collective operations as introduced in the 3.0 version of the MPI standard have also potential for overlap. This is an important asset, since especially the all-to-all collectives are a typical scalability bottleneck in supercomputing applications. The overlap availability of the non-blocking all-to-all data exchange as implemented in `MPI_Ialltoall` on Cray XC30 is presented in Figure 7 (right). The `MPI_Ialltoall` operation does not overlap properly, and some performance benefit from overlapping can be expected only with small message sizes and when using the Asynchronous Progress Engine.

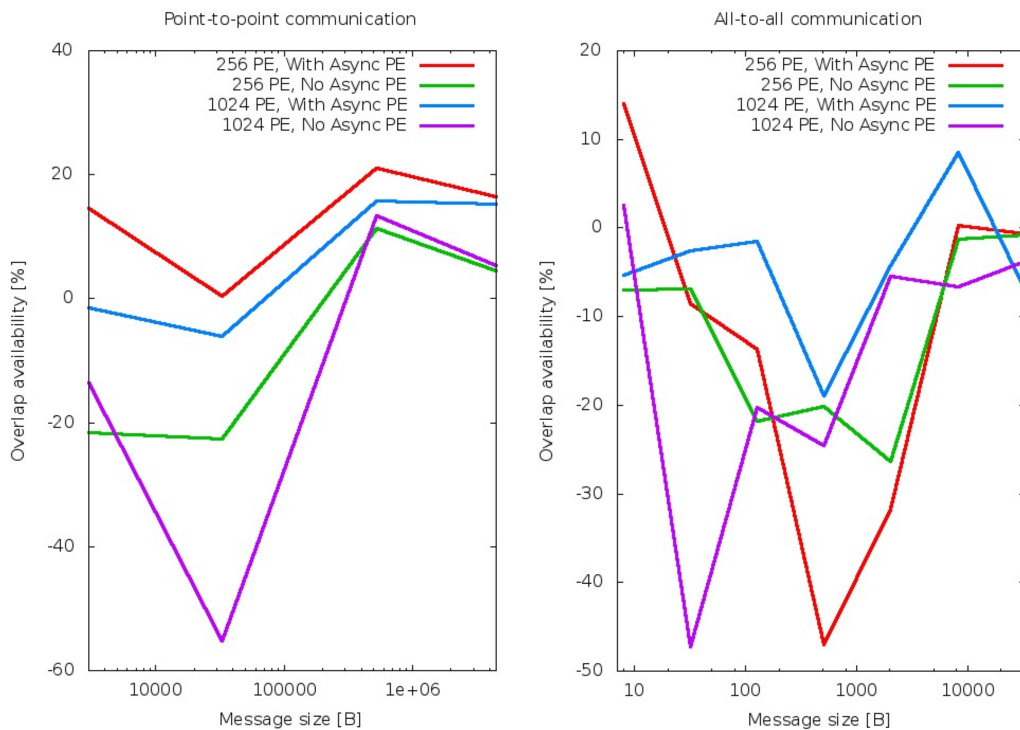

Figure 7: Overlap availabilities of non-blocking point-to-point and collective communication.

7 Micro-kernels

7.1 Motivation

7.1.1 Core-specialisation with Monolithic Kernel OS

On some HPC systems that employ a monolithic kernel operating system (such a standard Linux distribution, or a customised derivative of a standard Linux distribution), it is possible to separate the OS and the application processes on to different hardware processors. For example, in a Cray XC30, this feature is called core-specialisation. The user can reserve one (or more) processor core(s) for the OS kernel by inserting commands into the job-launch script. The application processes cannot use these reserved core(s), which means that more nodes will be needed to support the same number of application processes. However, the application processes should benefit from reduced OS jitter, i.e. reduced interference caused by the OS handling hardware interrupts, which requires context-switching and possibly disrupts memory caches and the TLB cache. A possible disadvantage of core-specialisation is that it introduces locality issues in NUMA systems. Passing a memory pointer from the application to the OS kernel may invalidate the associated cache entries in the application processor core when the OS kernel processor core accesses memory using that pointer. Some OS kernel functions and services should therefore benefit from co-location with the application processes that use them. A further possible disadvantage is that the reserved core may be overloaded and become a performance bottleneck if OS system calls are made by application processes too frequently or if hardware interrupts occur too frequently.

The disadvantages of core-specialisation suggest that a better option would be to split the OS into separate services that execute in different physical locations within each node (to address locality issues) and to replicate some OS services adding dynamic load-balancing between the replicas (to address bottle-neck issues).

7.1.2 Asynchronous Progress Engine with Non-Blocking MPI Communication

An asynchronous MPI progress engine, as described in Section 6, achieves the goals of core-specialisation for a single OS service, namely network I/O.

The intention is that the application is written to make efficient use of MPI non-blocking communication functions for all communication. When each non-blocking communication operation is started, the MPI library returns control to the application immediately. The MPI library progresses the communication in a separate thread or a separate process (the asynchronous progress engine). Meanwhile, the application does some useful computation work that does not depend on the result of the communication. At a later point, the application verifies that the communication operation is complete. When the asynchronous progress engine is scheduled on a dedicated processor core, this coding style overlaps communication and computation to achieve speed-up.

From the point-of-view of the operating system, this coding style means that the network I/O intensive portion of the application code (and therefore network I/O related system calls to the OS kernel) can be scheduled separately from the rest of the application. This allows one core to continue intensive computation, e.g. floating-point operations, whilst another core simultaneously handles the communication, i.e. network I/O, hardware interrupts from the network interface card and memory copies between user data buffers and network interface buffers.

One disadvantage of this approach is that it only benefits MPI communication. Other communications, such as direct socket connections to a client machine for interactive visualisation or steering of the application (as seen in HemeLB, for example), make network I/O system calls directly to the OS kernel from the application processor core rather than the processor core dedicated to the MPI progress engine.

This suggests that it may be better to extract the network I/O system calls from the OS kernel and create a separate network I/O service that can run on a dedicated processor core. In this manner, all network related system calls will be offloaded from the application processor cores to a dedicated processor core. This reduces the adverse effects of handling hardware interrupts generated by the network interface card(s); although it does require an efficient inter-process communication mechanism as all application system calls for network services necessitate at least one IPC message.

Further development of this idea indicates it may be beneficial to replicate the network I/O service process so that, for example, there is one such process for each physical network interface card in the node. At full line-speed data-rate, a single Gigabit Ethernet network connection with standard 1500-byte network packets requires a per-packet processing time no greater than 12μs. With multiple physical network interface cards operating at full line-speed data-rate, this processing time limit is proportionally reduced, e.g. 6μs for 2 NICs, 4μs for 3 NICs, etc. If each incoming packet on each NIC generates a hardware-interrupt then this per-packet processing time limit can be difficult to maintain. A common technique to mitigate this issue is interrupt coalescing, which reduces the frequency of interrupts by combining multiple network events when they occur with a short time-span of each other. An interrupt is then only generated when a certain number of network events have been combined or a timer expires. Inevitably, this results in a compromise between providing the lowest possible latency (which requires an immediate interrupt for each network event) and providing the highest possible bandwidth (which requires a limit on the maximum frequency of interrupts). Similar arguments apply to other network interconnection fabrics, such as Infiniband. Potentially, replicating the network I/O service module so that each replica is scheduled on a different processor core and deals with a single NIC would permit a lower latency without compromising the maximum achievable bandwidth even with multiple NICs per node.

7.1.3 Motivating Scenario for Separate OS Module for Disk I/O

The creation of node-local I/O servers achieves the goals of core-specialisation for another OS service, namely disk I/O.

The general concept of offload servers and the special case of I/O servers are described in Section 5. In that Section, it is envisioned that the offload or I/O servers are implemented as separate processes or threads in user-space, i.e. not within the OS kernel. However, the disk I/O service is ultimately provided by the operating system, usually as a system-level driver. The option of creating and exposing a new API for I/O servers is explored in Section 5.3 as a demonstration of a new general API proposed for all offload servers. This Section examines an alternative option: implementing I/O servers by transparently intercepting I/O system calls.

In general, (non-volatile) disks operate much slower than other (non-volatile) storage hardware, such as main memory, and much slower than computational hardware, such as a CPU or GPU. If application processes wait for disk I/O then overall performance is likely to very poor. Therefore disk I/O must be exposed to HPC applications via a non-blocking API and must be implemented in a manner that allows overlap with application computation.

For current operating systems, there are two approaches for non-blocking disk I/O: the data from all write operations is buffered in the kernel (e.g. normal POSIX I/O in Linux) or a non-blocking API is provided by the operating system (e.g. “OVERLAPPED” I/O in Windows). In both cases the data is actually written to disk after the I/O system call has completed and returned control to the application. This is normally implemented via a separate kernel thread or process, which inherently suits the design principles for a microkernel.

The messaging interface of a module in a microkernel operating system is very well-defined and completely isolates that module from the all other modules in the system. This should make interception of I/O system calls simpler than for a monolithic kernel.

The possibility of replicating the disk I/O module, e.g. one instance for each physical disk, is unlikely to increase performance because disk hardware is so slow. However, there is potential benefit to co-locating the disk I/O module with one or more network I/O modules in order to aggregate data within a small subset of (possibly remote) nodes before writing data to disk.

7.1.4 Motivating Scenario for Locality-Aware Memory Management OS Module

A well-written HPC program should minimise the frequency of memory allocation system calls during the majority of its execution. However, for some codes the amount of memory needed by each processor core will vary throughout the execution of the program in response to the needs of the algorithm and the distribution of data and work. In systems with Non-Uniform Memory Access (NUMA), memory allocation and management by the operating system must be locality-aware in order to make best use of fast cache memory. Even when no new memory is allocated, the access-pattern for existing memory allocations may change, for example to achieve load-balance between processor cores.

Memory pages can be migrated by the operating system to make them local to the processor core that is currently most frequently accessing them. A page-fault exception is generated whenever a processor core requests a memory page that is not present in locally accessible memory (either in processor caches or in the portion of main memory that is directly connected to the processor). The memory management module in the operating system handles the exception by interrupting the application, fetching the memory page (possibly involving communication with other processors) and then re-starting the application. Therefore, memory management is an OS service that requires locality-awareness and cannot usefully be centralised to a single processor core.

7.2 Methods

7.2.1 Customisation of a Standard Monolithic Kernel

A standard monolithic kernel operating system is usually split into many separate kernel processes (sometimes also called system daemons or background services), with each kernel process performing a different task or providing a different service. This provides an opportunity to move certain operating system services to dedicated hardware simply by influencing the scheduling decisions made by the operating system. Most operating systems support the concept of ‘affinity’ for all processes – effectively this is done by specifying a list (called a mask) of processor cores on which the process is allowed to be scheduled. Setting this affinity list, or mask, to a single processor core for a particular process (e.g. the network I/O system daemon) prevents the operating system from moving that process to another processor core during execution. Setting this affinity list to all cores except one for other processes (e.g. application processes) prevents those processes from interfering with the exceptional processor core.

The major advantage of this approach is that changes to the standard OS kernel are restricted to code that handles scheduling decisions: the affinity mask for system processes must be set to include only the reserved processor core(s) and the affinity mask for application processes must exclude the reserved processor core(s). A disadvantage is that this simple approach only works if the OS kernel functionality is already a separate process in the standard OS. This may be true for services such as network I/O but is less likely to be true for memory management. Even for OS services that are primarily handled by separate system processes, some of the functionality is likely to rely on central OS kernel functions. For example, if the network I/O system daemon process is forced to execute on a different processor core to a system process that provides dependent functionality, this may increase the amount of inter-processor communication and reduce the overall effectiveness of using dedicated hardware.

7.2.2 Micro-kernel Operating Systems

The defining characteristic of a micro-kernel operating system is that it is as small as possible whilst still being functional. All non-essential functionality is removed from the

operating system and is instead provided by non-OS processes. Figure 8 shows a schematic representation of a typical monolithic and a typical micro-kernel. The essential functionality, which remains in the operating system microkernel, includes an efficient mechanism for inter-process communication and some low-level memory management and process (or thread) scheduling ability. The non-OS processes that provide OS-like functionality are typically modules that are as self-contained as possible but may nevertheless depend on other non-OS processes. Thus, each process can be scheduled independently of all the others with little or no impact on performance due to frequent interaction. Commonly in a microkernel operating system, those modules that are not required are not loaded into memory reducing the memory usage compared to a monolithic kernel.

The OS services provided by a microkernel can often easily accommodate replicated or hierarchical modules. All dependencies between processes (whether application processes or system service processes) require explicit inter-process communication via the microkernel itself. This enables the re-direction of requests to the correct module or sub-module on a per-request basis. For example, two network I/O service processes may exist in a particular node, one for each physical network interface card installed in the node. When a request is made to communicate across the network, the inter-process message can be directed to the network I/O process that controls the appropriate hardware resource for that request.

One disadvantage of microkernels is that they generally do not provide a familiar programming interface (such as the full set of Linux system calls or the POSIX process model) to applications. Another disadvantage is that the inter-process communication can become a performance issue. A “hybrid kernel” design re-absorbs some performance-critical components into the microkernel module of the operating system, as depicted in Figure 9. This compromises the programmability benefits in the low-level code for system services but can achieve a more practically useful OS. The Windows NT kernel and Mac OS X are considered to be hybrid operating systems.

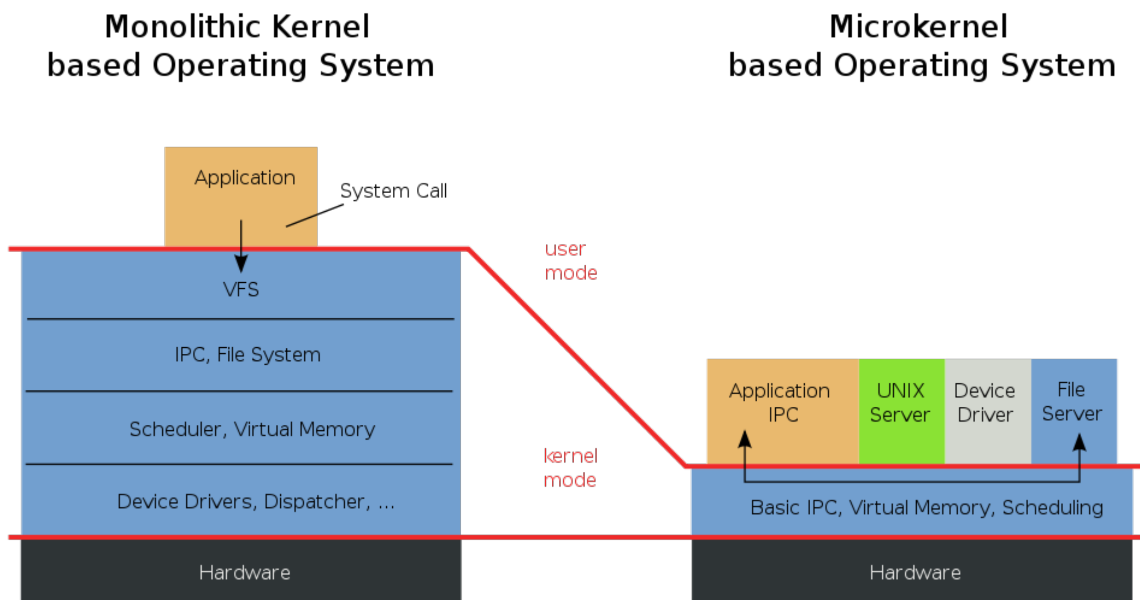

Figure 8: Schematic of operating system components in a typical monolithic kernel and microkernel [8]

© CRESTA Consortium Partners 2011

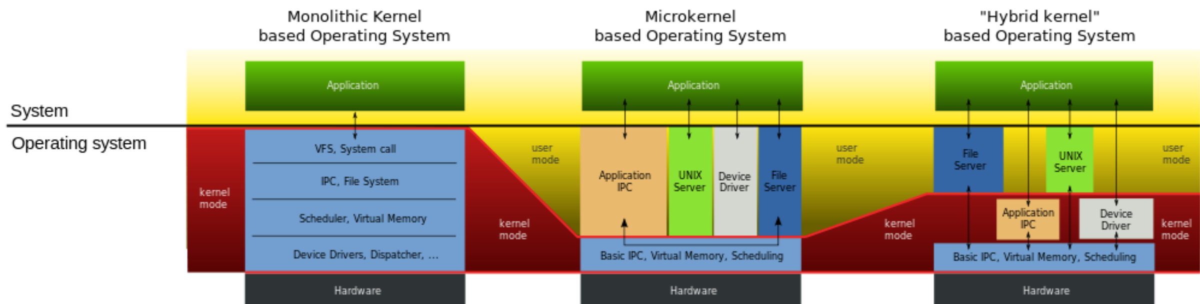

Figure 9: Comparison of Compartmentalisation in a Monolithic Kernel, a Microkernel and a Hybrid Kernel [9]

7.3 Conclusions

The microkernel concept has been a subject of practical research efforts for many decades. There have been some useful insights for operating system design and implementation. In particular, currently popular operating systems, such as Windows and Mac OS X, make use of the principles of microkernel design. However, there are significant difficulties for continuing efforts in this area. There is a significant knowledge hurdle requiring a steep learning curve for new participants in this research field. Furthermore, both in HPC and in the wider context of mass-market consumer systems, established operating systems are ubiquitous and entrenched in the mind-set of application programmers as well as the designers and purchasers of these systems. Therefore, new developments in the HPC OS space are often seen, and judged, as disruptive technologies.

Enhancing the ability of a “fat node” to accelerate a single application, or a small number of co-located applications, does not align well with mainstream requirements for consumer computers. A typical desktop or laptop computer must be more focused on fairly sharing available hardware resources to achieve high through-put despite over-subscription of those limited resources by dozens of disparate independent processes. An Exascale-capable method of efficiently exploiting fat nodes, within an increasingly restrictive power-budget, will necessarily diverge from the mainstream solutions available today and will probably become a niche product only used for a few of the largest machines in the world.

8 Future work

The research presented in this reports opens up a number of strands for future work. We have shown that co-location of HPC workloads is a feasible option for fat nodes, however there are still a number of hurdles to be overcome before this will be a generally applicable approach.

In terms of CRESTA, two different areas of investigation are relevant:

- **Power management**: we will continue our work on co-location of applications as part of Task 2.6.3 in CRESTA WP2. The basic idea is that using resources that would otherwise be idle should increase the scientific output without increasing power consumption significantly. Runtime is currently the most interesting metric for codes running on supercomputer architectures, as this is what is used in the charging models. However looking forward, users may instead be billed for the energy usage. The Cray XE6 has limited power measurement facilities but, with access to suitable counters on a future architecture, it will be interesting to revisit this study, redefining the co-location metric in terms of energy consumed rather than time elapsed and seeing whether this affects the conclusions drawn here. Many HPC systems have high base-line power consumption, even if all the cores are in their lowest sleep state; activating the cores does increase the power drawn, but the increase is small compared to the base power consumption. With such hardware, minimising energy consumption is exactly linked to minimising application runtime. On future HPC architectures, however, the base power requirement will be much lower and in this case there is more flexibility in constructing a co-location metric that reflects improved energy consumption and not minimised runtime.
- **Ensemble modelling and coupled simulations**: for co-design vehicles such as GROMACS, IFS and HemeLB, co-location could be a viable way to improve resource usage when running very large ensemble models or coupled simulations. More work is however needed in order to facilitate launching multiple binaries on shared resources.

Offload servers, in particular I/O servers, can be integrated into I/O intensive applications and evaluated as an alternative to MPI-IO. It may be possible to do this directly in applications such as OpenFOAM or Elmfire, or via benchmarks, which simulate the I/O behaviour. Similarly, using spare cores to host the asynchronous progress threads for non-blocking MPI needs to be evaluated in the context of one of the co-design vehicles. Again, OpenFOAM is a good example here because it has the option to transparently switch between blocking and non-blocking MPI calls.

One area we have not investigated is the use of one-per-node offload servers to manage data transfers between the separate memory spaces of CPU and attached accelerator(s). (We have, however, given this some consideration based on our experience with accelerators.) The argument for doing this is that data transfers are a significant overhead in such hybrid calculations and that a typical way of using such hybrid nodes is currently to use one MPI rank per accelerator on the CPU, leaving many cores idle. With current hardware, it is not clear how useful this approach would be. For accelerator hardware or runtimes that support asynchronous data transfers (e.g. to Nvidia GPUs), this is probably not useful. The CPU initiates data transfers, but these operations are non-blocking. If an offload server were employed to manage data transfers, the overheads of communicating with the server would be at least as large as having the main "compute" core doing the data transfers itself. In addition, the latest Nvidia Kepler GPUs support multiple MPI ranks sharing a single GPU (via the so-called HyperQ and CUDA proxy facilities). This allows more of the CPU to be effectively used for computation.

For accelerators that do not allow asynchronous data transfers, it may make more sense to consider using an offload server on a "spare" core on each node/process to

manage data traffic to and from the CPU, whilst the "active" CPU cores are used for productive computation. Given the tightly coupled nature typical of applications using GPUs a threaded implementation would probably be required. We did not, however, have access to such hardware during the period of this study to pursue this.

Longer term, as we approach the Exascale, the industry trend appears to be towards closer integration of CPU and accelerator hardware. Even if the memory spaces are not completely unified, they will be much more closely linked. In the ARM Mali GPU, for instance, the memory space is shared between CPU and GPU but is not cache coherent, so data transfers are essentially reduced to cache flushes. As such, the overhead of data movement in a hybrid application becomes less onerous and it is not clear that an offload server model is then necessary. A related example is the Intel Xeon Phi coprocessor. Whilst this can be used as an offload accelerator, most users currently take the approach of running the entire application on the coprocessor and, again, there is no need for an offload server.

So whilst we will track the hardware development trends, we do not intend to pursue the use of offload servers for accelerator data transfers during the remainder of the CRESTA project.

© CRESTA Consortium Partners 2011

9 Conclusions

Current and projected trends in system architecture imply that fat nodes in HPC are here to stay. Single applications often cannot fully exploit these fat nodes because of limitations imposed by the memory and communications systems, and thus a number of cores on these fat nodes can sit idle. This deliverable has investigated a range of different ways in which any spare (or empty) cores on fat nodes can be exploited usefully, with an emphasis on improving the science throughput.

The approaches we have explored here range from co-locating HPC workloads on shared resources to offloading specific tasks to empty cores. Although some of these approaches are conceptually simple, their implementations on tightly coupled HPC systems are complex and, as highlighted in Section 8, more research in this area is needed. A positive side-effect of an improved exploitation of fat nodes may be that of increasing energy efficiency. Again, this issue needs to be explored in further research.

Overall, this deliverable describes how we can progress beyond the current state-of-the-art in the use of fat nodes and makes clear suggestions for how the different approaches can be used with the CRESTA co-design vehicles.

© CRESTA Consortium Partners 2011

10 References

[1] CRESTA deliverable D2.1.1 “Architectural developments towards Exascale”.

[2] Dean A. Klein, “The Future of Memory and Storage: Closing the Gap”. Presentation at the Windows Hardware Engineering Conference 2007, Los Angeles, USA. Talk slides available online at:
http://download.microsoft.com/download/a/f/d/afdfd50d-6eb9-425e-84e1-b4085a80e34e/sys-t308_wh07.pptx
[last accessed 27th August 2013]

[3] Koop, Matthew J., Miao Luo, and Dhabaleswar K. Panda. "Reducing network contention with mixed workloads on modern multicore clusters." *IEEE International Conference on Cluster Computing and Workshops, 2009. CLUSTER'09*. IEEE, 2009.

[4] Breslow, A. D. et al., "The Case for Co-location of HPC Workloads", Concurrency and Computation: Practice and Experience (in press), 2013.

[5] Joshua Mora, “Do theoretical FLOPs matter for real application's performance?”. Presentation at the HPC Advisory Council Conference 2012 in Malaga, Spain. Talk slides available online at:
http://www.hpcadvisorycouncil.com/events/2012/Spain-Workshop/pres/6_AMD.pdf
[last accessed 27th August 2013]

[6] P. Dagna: „OpenFOAM on BG/Q porting and performance“,
http://www.hpc.cineca.it/sites/default/files/4_OpenFOAM_on_BGQ_porting_and_performance_Dagna.pdf

[7] B. Lindi: “I/O-profiling with Darshan”,
http://www.prace-ri.eu/IMG/pdf/IO-profiling_with_Darshan-2.pdf

[8] From Wikipedia page: http://en.wikipedia.org/wiki/Microkernel URL:
http://upload.wikimedia.org/wikipedia/commons/thumb/6/67/OS-structure.svg/1000px-OS-structure.svg.png

[9] From Wikipedia page:
http://en.wikibooks.org/wiki/Operating_System_Design/Print_Version URL:
http://upload.wikimedia.org/wikipedia/commons/thumb/d/d0/OS-structure2.svg/1000px-OS-structure2.svg.png

© CRESTA Consortium Partners 2011