Optimising a Fluid Plasma Turbulence Simulation on Modern High Performance Computers

Thomas David Edwards

A thesis submitted in fulfilment of the requirements

for the degree of Doctor of Philosophy

to the

University of Edinburgh

2010

Declaration

I do hereby declare that this thesis was composed by myself and that the work described within is my own, except where explicitly stated otherwise.

Thomas David Edwards

March 2010

Acknowledgements

For Asha, whose love, encouragement and support made all this possible,

for my parents who gave me the joy of learning and knowledge,

and for my supervisors, Joachim and Peter who were always there and always understanding.

Abstract

Nuclear fusion offers the potential of almost limitless energy from sea water and lithium without the dangers of carbon emissions or long term radioactive waste. At the forefront of fusion technology are the tokamaks, toroidal magnetic confinement devices that contain miniature stars on Earth. Nuclei can only fuse by overcoming the strong electrostatic forces between them which requires high temperatures and pressures. The temperatures in a tokamak are so great that the Deuterium-Tritium fusion fuel forms a plasma which must be kept hot and under pressure to maintain the fusion reaction. Turbulence in the plasma causes disruption by transporting mass and energy away from this core, reducing the efficiency of the reaction. Understanding and controlling the mechanisms of plasma turbulence is key to building a fusion reactor capable of producing sustained output.

The extreme temperatures make detailed empirical observations difficult to acquire, so numerical simulations are used as an additional method of investigation. One numerical model used to study turbulence and diffusion is CENTORI, a direct two-fluid magneto-hydrodynamic simulation of a tokamak plasma developed by the Culham Centre for Fusion Energy (CCFE formerly UKAEA:Fusion). It simulates the entire tokamak plasma with realistic geometry, evolving bulk plasma quantities like pressure, density and temperature through millions of timesteps. This requires CENTORI to run in parallel on a Massively Parallel Processing (MPP) supercomputer to produce results in an acceptable time.

Any improvements in CENTORI's performance increases the rate and/or total number of results that can be obtained from access to supercomputer resources. This thesis presents the substantial effort to optimise CENTORI on the current generation of academic supercomputers. It investigates and reviews the properties of contemporary computer architectures then proposes, implements and executes a benchmark suite of CENTORI's fundamental kernels. The suite is used to compare the performance of three competing memory layouts of the primary vector data structure using a selection of compilers on a variety of computer architectures. The results show there is no optimal memory layout on all platforms so a flexible optimisation strategy was adopted to pursue "portable" optimisation i.e optimisations that can easily be added, adapted or removed from future platforms depending on their performance.

This required designing an interface to functions and datatypes that separate CENTORI's fundamental algorithms from repetitive, low-level implementation details. This approach offered multiple benefits including: the clearer representation of CENTORI's core equations as mathematical expressions in Fortran source code allows rapid prototyping and development of new features; the reduction in the total data volume by a factor of three reduces the amount of data transferred over the memory bus to almost a third; and the reduction in the number of intense floating point kernels reduces the effort of optimising the application on new platforms.

The project proceeds to rewrite CENTORI using the new Application Programming Interface (API) and evaluates two optimised implementations. The first is a traditional library implementation that uses hand optimised subroutines to implement the library functions. The second uses a dynamic optimisation engine to perform automatic stripmining to improve the performance of the memory hierarchy. The automatic stripmining implementation uses lazy evaluation to delay calculations until absolutely necessary, allowing it to identify temporary data structures and minimise them for optimal cache use. This novel technique is combined with highly optimised implementations of the kernel operations and optimised parallel communication routines to produce a significant improvement in CENTORI's performance. The maximum measured speed up of the optimised versions over the original code was 3.4 times on 128 processors on HPCx, 2.8 times on 1024 processors on HECToR and 2.3 times on 256 processors on HPC-FF.

Contents

Chapter 1

An Introduction to Fusion Energy and CENTORI

Fears of rapid and irreversible climate change and the need to meet the increasing energy demands of the developing world make finding alternatives to fossil fuels one of the most important tasks in science and engineering of the 21st Century[1, 2, 3]. Power sources that do not emit carbon dioxide will be a vital part of any future energy strategy and while technologies exist to generate energy without producing CO_2 each has its own shortcomings:

- *Renewable energy* like wind, wave and solar power are proven real world technologies and their use has been expanded rapidly recently. As a power source they have relatively low energy densities and there are concerns about the consistency of energy produced which limits their potential[4, 5].
- *Nuclear fission* reactors are capable of meeting the base load without emitting any CO_2 and construction of a new generation is also being considered after a long moratorium[5]. However, questions remain about the long term disposal of radioactive waste and ensuring the security nuclear material and technology[6].
- *Bio-fuels* have great potential to replace traditional hydrocarbons; however concerns remain about the extent of their carbon neutrality and the impact of taking large swathes of agricultural land away from food production on the global economy[7].

Nuclear fusion is another promising alternative technology that has advantages over many other forms of energy production as

- it only has civilian applications with no risk of meltdown or catastrophic accidents,
- there are no carbon emissions from energy production,
- there are vast supplies of fuel on Earth[8],
- and it only produces small volumes of short-term radioactive waste[9, 10, 11].

Yet while the potential of fusion power has been recognised since the 1950s, realisation of the technology has proved elusive and a commercial fusion power station remains several decades away.

1.1 Potential Fusion Technologies

The viability of the fusion reaction has been proven: the Sun is a giant fusion reactor and successful fusion experiments have been performed on Earth. Creating a sustained and controlled reaction that could be used in a power station has proven more difficult. There are multiple research topics currently being investigated by the fusion community around the world.

1.1.1 The D-T Reaction

Most fusion research focuses on the reaction between two hydrogen isotopes, deuterium (D or ^{2}H) and tritium (T or ^{3}H) as this has a large cross section (probability of interaction), 5.0 barns, and produces a large amount of energy. The reaction is

$$^2\mathrm{H} + {}^3\mathrm{H} \longrightarrow {}^4\mathrm{He}(3.5\mathrm{MeV}) + \mathrm{n}(14.1\mathrm{MeV})$$

The majority of the energy is released in the high energy neutrons (75%) which have no charge and so cannot be deflected by magnetic or electrostatic fields. The remaining energy is passed to the helium nucleus. Forcing two nuclei together, against the electrostatic repulsion, until the strong nuclear force can bind them together requires a certain amount of energy, the Coulomb barrier. When the kinetic energy of two colliding nuclei is sufficient to overcome the barrier nuclear fusion will occur. The temperatures where particles have sufficient kinetic energy to regularly overcome the Coulomb barrier are so high that matter forms a plasma. Plasmas are a fourth state of matter where electrons are stripped from atoms, forming a fluid of ions and electrons. Though there maybe charge separations on the smallest scales, on larger

scales the plasma is quasi-neutral, i.e. there are equal amounts of positive and negative charge in any given volume.

In 1955, J.D Lawson derived the criterion necessary (but not sufficient) for a D-T fusion reaction to become self sustaining[12]. The rate of fusion reactions is determined by three quantities: n_e, the electron density, T the plasma temperature, and τ_E, the confinement time which is a measure of the rate at which energy is lost from the system. The Lawson criterion states that this product must be above the threshold value of:

$$n_e T \tau_E > 3 \times 10^{28} \mathrm{Km}^{-3}\mathrm{s}$$

1.1.2 Sources of Deuterium and Tritium Fuel

Deuterium is a natural isotope of hydrogen, making up 0.015% by number, that can be refined from seawater. Tritium is radioactive, with a half life of 12.32 years, which means there are no natural sources, though it can be bred from lithium which has significant reserves in the Earth's crust and seawater[8, 13]. The D-T fusion reaction will be a good source of high energy neutrons and so it is expected that tritium can be manufactured by lining the walls of any reactor with a lithium blanket to undergo the following reaction:

$$\begin{aligned} {}^{7}\mathrm{Li} + \mathrm{n} &\rightarrow {}^{4}\mathrm{He} + {}^{3}\mathrm{H} + \mathrm{n} + 2.5\mathrm{MeV} \\ {}^{6}\mathrm{Li} + \mathrm{n} &\rightarrow {}^{4}\mathrm{He} + {}^{3}\mathrm{H} + 4.5\mathrm{MeV} \end{aligned}$$

1.1.3 Confinement

The rate of fusion reactions is determined by the density and temperature of the plasma and while creating conditions that are hot and dense enough for fusion are a necessary condition, the plasma has to be confined for a sufficient length of time.

Gravitational Confinement

Gravity is a natural force capable of confining matter at sufficient densities that nuclear fusion will occur. However the force of gravity is so weak that only extremely massive bodies like stars have sufficient gravitational force to sustain a fusion reaction. Though this form of nuclear fusion is important in astrophysics it is not a viable method of generating energy on earth.

Inertial Confinement

Another method to create and sustain the high temperatures and pressure required for a fusion reaction is to rapidly implode a body of fusion fuel using a controlled explosion or rapid heating. This technique has been successfully proven on earth in "H-Bombs" that use a nuclear fission device to ignite the fusion reaction. More controlled techniques use a high powered laser to uniformly heat the surface of a spherical pellet of Deuterium-Tritium fuel causing a shrinking shock wave to implode to a single point at the centre. This will cause fusion to occur and create a chain reaction from the centre to the edge[14].

The ultimate success of the technique depends on the total power of the laser, its efficiency and the ability to deliver the energy precisely and uniformly over the spherical pellet's surface. The gain in energy from the fusion reaction must be sufficiently large to cover the costs of powering the laser which has only become feasible with advances in laser technology. The flagship National Ignition Facility at Lawrence Livermore National Laboratory intends to use the world's largest laser (500 TW) to deliver 2 MJ to the surface of the pellet in 192 separate beams and experiments designed to produce full ignition commenced in January 2010[15].

Magnetic Confinement

The temperature of a burning fusion plasma is so great that the plasma would instantly vaporise any material with which it came into contact. Inertial confinement uses only tiny pellets of fuel, for larger amounts a different form of confinement is required. Plasma is an electrically conducting fluid of charged particles and so can be directed using magnetic fields. Since the earliest days of civilian fusion research many designs of "magnetic bottle" have been used, however the most successful design has been the tokamak, developed by Soviet scientists in the 1960s.

Tokamaks are large toroidal vacuum vessels which confine the plasma using strong magnetic fields. Rather than relying on high densities the plasma is heated to temperatures beyond the core of the sun to initiate fusion. Energy is delivered from the system through the emission of neutrons which are unaffected by magnetic field lines whilst the helium ash is retained in the plasma. By lining the walls of the tokamak with a lithium blanket, the high energy neutrons could be used to generate the Tritium fuel required as explained in Section 1.1.2.

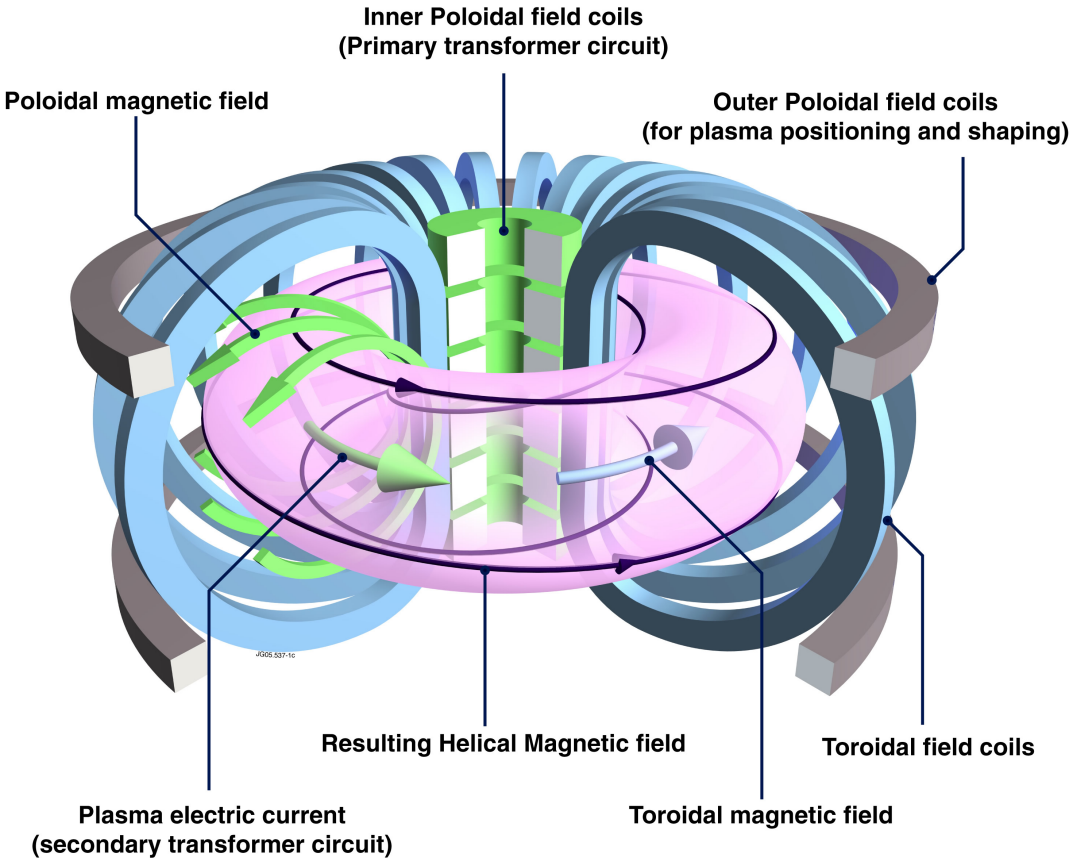


Figure 1.1: Diagram illustrating the tokamak principle: arrangement of magnetic field coils and the resulting magnetic field that confines the plasma. Image: EFDA-JET

1.1.4 Tokamaks

Tokamaks are the most common magnetic confinement device in fusion research reactors. In their simplest design they comprise a toroidal vacuum vessel which has a set of powerful electromagnetic coils producing a toroidal magnetic field. The plasma current around the torus forms a poloidal field, resulting in helical field lines around the torus which the charged particles follow through cyclotron motion. Figure 1.1 shows an overview of the magnetic field coils and resulting magnetic field that confines the plasma.

In the majority of operating tokamaks, the coils generating the required magnetics fields are made from copper, which is cheaper and easier than using superconducting coils though it results in a large amount of energy being used to generate the fields and being lost through resistance. Some current tokamak designs like EAST and KSTAR, and important future designs like ITER use superconductors to generate the fields to avoid problems with resistance.

1.1.5 Tokamak Experiments

For many years, the foremost experimental tokamak for fusion physics has been the Joint European Torus (JET) at Culham Science Centre, Oxfordshire, United Kingdom. It is currently the largest tokamak in the world, having been operational since 1983. It has undergone several extensive upgrades and refits as more has been learnt and is fully equipped for handling Tritium. It was the first tokamak to produce a significant amount of fusion power and in 1997 during a Deuterium - Tritium campaign produced 16MW of power[16]. Currently it is being used to refine the design for the International Thermonuclear Experimental Reactor (ITER) which is being built in Cadarache in France. ITER is a multi-national effort that will build upon what has been learnt from JET and other fusion research programs "to demonstrate the scientific and technological feasibility of fusion power for peaceful purposes"[17].

The Mega-amp Spherical Tokamak (MAST)[18] is a device built and operated by the Culham Centre for Fusion Energy based at the same site in Culham. Though smaller than JET, the MAST program has a different design which has provided valuable results which are being fed back into the design of ITER. MAST follows on from its predecessor, the Small Tight Aspect Ratio Tokamak (START), by having a tighter aspect ratio (the ratio of the major radius to the minor radius of the torus) than other tokamaks. This smaller aspect ratio means devices are smaller and research suggests that they may be more efficient by reducing the scale of the toroidal magnetic field required for plasma stability.

Tokamaks are at the forefront of current magnetic confinement fusion research and are believed to be the most likely magnetic confinement technology to first produce a viable power station prototype[19].

1.2 Turbulence and Transport in the Tokamak

To reach or surpass the Lawson criterion the plasma in the tokamak must reach the highest temperature and density possible and be confined for as long as possible. Much research on tokamaks has focused on creating and maintaining optimal temperatures and density profiles in the plasma to produce the most efficient reactions. Transport and diffusion of species and energy from the central core of the tokamak is one of the major obstacles to obtaining the longest confinement times. A large amount of effort has been expended in studying the mechanisms of transport in tokamak plasmas.

1.2.1 Mechanisms of transport

Theoretical plasmas without resistance are perfectly contained by the magnetic flux surfaces forming perfectly sealed systems. However, these simple models do not consider the interaction of individual charged particles through Coulomb collisions.

Classical transport

$$r_L = \frac{mv_\perp}{eB} \tag{1.1}$$

Charged particles in a magnetised plasma move in a helix around magnetic field lines. The radius of this orbit, r_L the Larmor radius, is related to the mass of the species, m, the magnitude of the magnetic field B and the speed of the particle in the direction perpendicular to the magnetic field, $v_\perp$, by Equation 1.1. The Larmor radius can be quite large in a tokamak plasma (of the order of centimetres for ions) but the averaged resulting motion has a guiding centre which moves along the magnetic field, $\mathbf{B}$. Particles that move along parallel magnetic field lines will occasionally collide, knocking one another across the flux surfaces causing a net flow of energy and particles across the magnetic flux surfaces[20]. The individual particles are subject to a random walk which is modelled as a diffusion coefficient, D, which is proportional to $D \propto \nu_c r_L^2$, where ν_c is the rate of collisions. As ions have a much larger Larmor radius than electrons, their diffusion rate will be much larger, however the separation charge caused by unequal diffusion of particles generates an electric field which limits the diffusion to the lower electron diffusion rate. This is classical transport which is unavoidable as it is a consequence of the plasma's structure.

Neoclassical Transport

Other collisional effects arise from inhomogeneous properties of a tokamak, the toroidal shape of the tokamak and the resulting non-uniform magnetic field strength that can trap particles on the outboard side in a magnetic mirror, to produce so called banana orbits[21]. Much work has been done to evaluate these effects on bulk transport and can be accurately modelled using *neoclassical* transport models. Combined with classical transport, neoclassical effects are also unavoidable and represent the lower bound on the amount of diffusion occurring in a tokamak plasma.

Anomalous Transport

The rate of energy loss measured in experiments is generally much greater than that predicted by classical and neoclassical diffusion models which implies there are other, so called anomalous, mechanisms of transport operating on the plasma. This energy loss is expected to occur from micro instabilities which are reliant on kinetic effects and the interaction of the two fluids. These instabilities form and interact as electromagnetic fluctuations which have positive growth rates and are driven by the large gradients in the plasma. These effects contribute to turbulence in the plasma which transports particles and energy across flux surfaces that is ultimately lost.

Turbulence is a widely studied phenomenon in many fields[22][23] and the cause and mechanisms of turbulent transport in plasmas are an important field in fusion research. The results have provided insight into better ways to control the current generation of tokamaks and has influenced the design of the next generation.

1.3 Modelling Tokamak Plasmas

Analytical studies of fusion plasmas have provided many insights into transport and turbulence, however being nonlinear phenomena they resist a complete solution. Experimentalists attempt to identify and measure the structures and features observed during their research on tokamaks, but the extreme conditions limit the detail of diagnostic results. A third approach is numerical simulation, which has become an increasingly important technique in modern science, allowing mathematical models to be studied in detail and directly compared against experimental observations.

It is only in the last few decades that sufficient computational resources became available to allow scientists to run models of sufficient resolution and sophistication. These new tools are used extensively in nearly all areas of modern science increasing insight into phenomena that are difficult, expensive or impossible to study by other means. There is the ultimate caveat that simulation does not replace direct observation and it should be kept in mind that they are ways of exploring the properties of models rather than actual investigations of real world phenomena.

There are many different models of plasmas used in simulations of plasmas, each having its own advantages and disadvantages when compared. Most research groups will use a suite of tools to investigate phenomena. Some of the more commonly used models in plasma turbulence

research are discussed here:

1.3.1 Gyro-kinetic Models

Kinetic theory is a fundamental description of plasma properties by modelling the distribution of velocity and position of each species in phase space. The Maxwell equations are applied to account for the effect of charge densities on the distribution functions and allow the modelling of short wavelength phenomena which are potentially smaller than the Larmor radius. This in principle requires models to track each species in six dimensions, however the frequency of individual particles around the gyro-centre is so rapid it can be averaged out which leads to a 5 dimensional model. Codes using gyrokinetic models include GS/2[24, 25] and GENE[26].

1.3.2 Fluid Models

Fluid models assume that the plasma being simulated is sufficiently collisional that the distribution of particles and can be considered locally a Maxwellian distribution. From this assumption the plasma can be considered a continuum and plasma moments can be rigorously derived for bulk properties like pressure p, fluid flow v and temperature T for each of the species involved. By using approximations, as opposed to full kinetic theory, the equations that relate the bulk quantities require less computation to solve. Magnetohydrodynamics (MHD) is a combination of the Navier-Stokes Equations for general fluids and Maxwell's Equations for electromagnetism. The Ideal MHD descriptions assume that the plasma is perfectly conducting whereas more sophisticated descriptions include additional terms to deal with resistance and diffusion in the plasma.

1.4 An Overview of CENTORI

Magnetohydrodynamics successfully describes many large-scale phenomena and has been used extensively to study reactor plasma turbulence. Extensions to the general equations attempt to parametrise processes that occur below the grid scale, like diffusion and transport, to improve the overall representation of these physical processes.

CUTIE[27] is a Large Eddy Simulation of reactor plasma by A. Thyagaraja of the Culham Centre for Fusion Energy (CCFE). It is an extended implementation of the magnetohydrodynamic model that evolves the two fluids in a cylindrical coordinate system with periodic

boundary conditions. Physical parametrisations attempt to account for many features that are absent in the core equations like the toroidal geometry and neoclassical transport. As CUTIE is a fluid model it is capable of simulating the entire tokamak on a single computer and has been very successful at qualitatively describing phenomena seen in experimental observation.

CENTORI is a two-fluid magnetohydrodynamic continuum code that evolves bulk plasma properties like temperature, density and pressure for each of the fluids, the ions and the electrons. It was developed as the successor to CUTIE by P. Knight (also CCFE) and A. Thyagaraja as an original code and uses a more sophisticated physical model that fully incorporates the toroidal geometry of the tokamak. It is also a fluid model and so is capable of simulating phenomena from the meso to macro scales over the entire tokamak. It also includes a suite of parametrisations that model sub-grid scale effects. Though CENTORI is not designed to deal with important plasma phenomena like edge physics it can be used to simulate plasmas in a wide range of tokamaks like conventional aspect ratio tokamaks ITER and JET as well as tight aspect ratio spherical tokamaks like MAST.

Due to the increased complexity of the physical model and geometry changes CENTORI is more computationally demanding than CUTIE. The scientific and algorithmic model places a requirement that time steps be of the order of 0.1 ns though confinement times are typically in the order of many milliseconds. This requires tens of millions of computationally intensive time steps to evolve the simulation completely. To reach the solution in an acceptable time CENTORI was adapted to run in parallel using the industry standard Message Passing Interface (MPI)[28]. Running in parallel offers significant speed advantages over running on a single processor but requires access to supercomputers that form Massively Parallel Processors (MPP).

1.4.1 CENTORI's fundamental equations

At the time of writing the core model of CENTORI is described by the equations in this section, though CENTORI is being continually refined and development as an ongoing research project. These equations are extensions to the resistive MHD equations with additional terms added for extra physical processes not described in ideal MHD, and forcing which is external to the system (i.e. Neutral Beam Injection, RF Heating etc). All equations are presented in the Gaussian unit system and a full description of the mathematical derivation of the CENTORI's fundamental algorithms is available in the CENTORI Source Code Documentation by P. Knight[29]. The scientific justification and derivation of the additional terms is beyond the scope of this thesis

and will be dealt with by future scientific publications relating to CENTORI by its original authors. The equations are presented for completeness and to give some insight into the nature of the algorithm being solved computationally in code.

Prognostics Quantities

$\mathbf{v_i} \equiv$ ion velocity (cm/s)

$\mathbf{v}_e \equiv$ electron velocity (cm/s)

$\mathbf{A} \equiv$ vector potential (statvolt)

$n_e \equiv$ electron number density (cm^{-3})

$(= n_i$, ion number density, via quasi-neutrality)

$t_i \equiv$ ion temperature (erg)

$t_e \equiv$ electron temperature (erg)

Auxiliary Quantities

$\Phi \equiv$ electric (scalar) potential (statvolt)

$\mathbf{B} \equiv$ total magnetic field (Gauss)

$\mathbf{J} \equiv$ total current density ($\mathrm{statamp/cm}^2$)

$\mathbf{E} \equiv$ total electric field (statvolt/cm)

$p_i = n_e\, t_i \quad \equiv$ ion pressure ($\mathrm{erg/cm}^3$)

$p_e = n_e\, t_e \quad \equiv$ electron pressure ($\mathrm{erg/cm}^3$)

Momentum Equation

The momentum equation, or equation of motion, describes the force balance conditions:

$$
\begin{aligned}
m_i n_e \left(\frac{\partial \mathbf{v_i}}{\partial t} + \mathbf{W} \times \mathbf{v_i} \right) &= \frac{\mathbf{J} \times \mathbf{B}}{c} - \nabla(p_i + p_e) - \frac{m_i n_e}{2} \nabla(\mathbf{v_i}.\mathbf{v_i}) - m_i n_e \chi_v (\nabla \times \mathbf{W}) + \mathbf{S_v} \\
\mathbf{W} &= \nabla \times \mathbf{v_i} \quad \equiv \text{vorticity } (\mathrm{s}^{-1}) \\
\chi_v &\equiv \text{velocity diffusivity } (\mathrm{cm}^2/\mathrm{s}) \\
\mathbf{S_v} &\equiv \text{external force } (\mathrm{dyne}/\mathrm{cm}^3) \\
c &\equiv \text{speed of light } (3 \times 10^{10} \mathrm{cm/s}) \\
m_i &\equiv \text{ion mass (g)}
\end{aligned}
$$

which uses the vector identity $\frac{1}{2}\nabla(\mathbf{A}.\mathbf{A}) + (\nabla \times \mathbf{A}) \times \mathbf{A} = (\mathbf{A}.\nabla)\mathbf{A}$ to replace the term $(\mathbf{v_i}.\nabla)\mathbf{v_i}$ in standard MHD with $\mathbf{W} \times \mathbf{v_i}$ and $\frac{1}{2}\nabla(\mathbf{v_i}.\mathbf{v_i})$. The fourth term is added by CENTORI to represent additional diffusion in the system (including neoclassical diffusion). The fifth term accounts for an external source of momentum to the plasma (e.g. RF heating).

Faraday's Law

This is the first of three of Maxwell's equations used explicitly by CENTORI, however CENTORI evolves the magnetic potential rather than the **B** field.

$$
\begin{aligned}
\frac{1}{c}\frac{\partial \mathbf{B}}{\partial t} &= -\nabla \times \mathbf{E} \\
\text{Equivalently, } \frac{1}{c}\frac{\partial \mathbf{A}}{\partial t} &= -\mathbf{E} - \nabla \Phi
\end{aligned}
$$

Equations of State

In the absence of energy sources or sinks,

$$
\begin{aligned}
\frac{d}{dt}\left(\frac{p_i}{\rho_m^{\gamma}}\right) &= 0 \\
\frac{d}{dt}\left(\frac{p_e}{\rho_{m,e}^{\gamma}}\right) &= 0 \\
\text{where } \frac{d}{dt} &\equiv \frac{\partial}{\partial t} + \mathbf{v_i}.\nabla \text{ (convective time derivative)} \\
\gamma &= \frac{5}{3} \text{ ratio of specific heats} \\
\rho_{m,e} &\equiv \text{electron mass density } (\mathrm{g}/\mathrm{cm}^3)
\end{aligned}
$$

Mass Continuity

This equation expresses the conservation of mass. An equivalent equation holds for the electrons. This version allows for particles to be added from an external source (representing neutral beam injection etc).

$$\begin{aligned} m_i \left(\frac{\partial n_e}{\partial t} + \nabla.(n_e \mathbf{v_i}) \right) &= S_n \\ S_n &\equiv \text{particle source rate (g cm}^{-3}\text{s}^{-1}) \\ m_i &\equiv \text{ion mass (g)} \end{aligned}$$

Divergence of B

Maxwell's equations include the following well-known relationship:

$$\nabla.\mathbf{B} = 0$$

which is guaranteed as a consequence of $\mathbf{B} = \nabla \times \mathbf{A}$.

Ampère's Law

$$\mathbf{J} = \frac{c}{4\pi} \nabla \times \mathbf{B}$$

Ohm's Law

$$\begin{aligned} \mathbf{E} &= -\frac{\mathbf{v_i} \times \mathbf{B}}{c} + \frac{\mathbf{J} \times \mathbf{B}}{e\, n_e\, c} + \eta \mathbf{J} - \frac{\nabla p_e}{e\, n_e} \\ \text{where } \eta &\equiv \text{resistivity (s)} \\ e &\equiv \text{electron charge } (4.8 \times 10^{-10} \text{statcoulomb}) \end{aligned}$$

The first and second terms on the right hand side are effectively the Hall effect, the third due to resistance. The fourth term is an extension to the standard equations due to the pressure gradient. The plasma's resistivity η is modelled using the Spitzer resistivity modified by a neoclassical estimate.

1.4.2 Discretisation and Normalisation

CENTORI solves these equations numerically by iteratively evolving the primary quantities on a structured finite mesh of points in space for a discrete time interval (time step). Rather than solving the quantities exactly as defined they are normalised and separated into the mean or equilibrium component and the fluctuating component. This simplifies the code that implements the solver algorithm and allows quantities to be averaged over the mean flux surfaces defined by the equilibrium state. Many other important physical quantities are calculated incidentally as part of the algorithm and form part of the diagnostic output. Full details of the discretisation, normalisation and derived quantities are part of CENTORI's core Source Code Documentation[29].

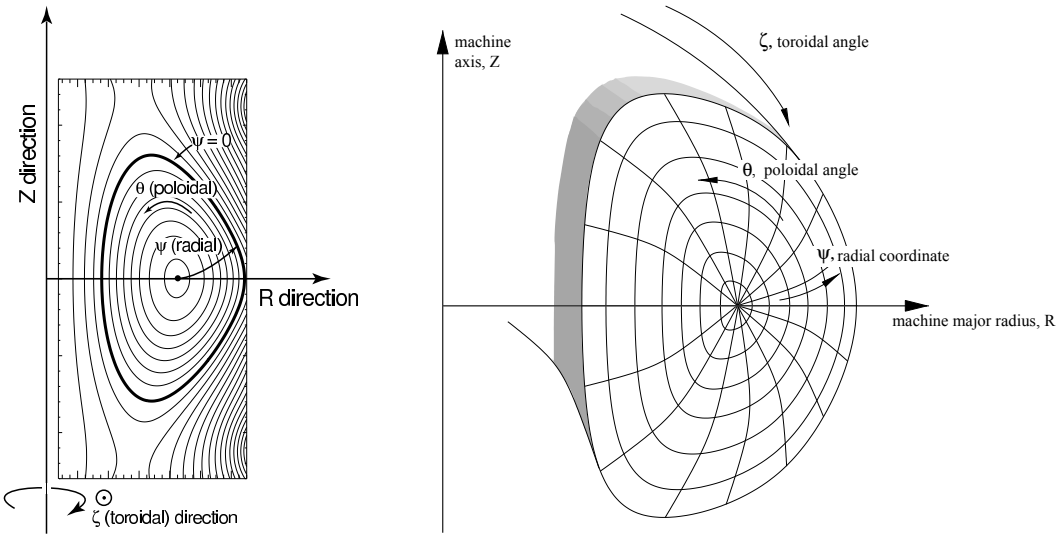
1.4.3 Defining the coordinate system

Tokamaks are toroidal devices so using a toroidal coordinate system simplifies the description of the model. The coordinates consist of two angular coordinates, θ and ζ, and a linear coordinate ψ. Figure 1.2a shows the angular toroidal coordinate, ζ, and the angular poloidal coordinate, θ, whereas the third coordinate, ψ, is derived from the equilibrium magnetic field. ψ is the poloidal flux function and is independent of ζ, it has a minimum value at the magnetic axis which forms a circle in the toroidal direction. Lines of constant ψ form flux surfaces which are a nested set of toroidal structures illustrated in Figure 1.2a. They are important features in tokamak physics because many physical parameters are effectively constants on the surfaces because the plasma flows along the surfaces are very rapid, quickly dissipating any local fluctuations.

The covariant, contravariant and physical basis sets

The coordinates (ψ, θ, ζ) form a right handed, non-orthogonal coordinate system. As the system is non-orthogonal two other basis sets are defined, the dual basis sets of the covariant and contravariant. Most binary operations can only be performed on vectors expressed in the same basis set while unary finite difference operators require vector data in either the contravariant or covariant basis sets, producing output in the same or opposite basis.

Vectors values cannot be expressed on the central magnetic axis, as the basis here is degenerate at this point because it is impossible to define $\nabla\theta$, therefore any values have to be expressed in the laboratory coordinate system used during initialisation.



(a) Plot showing a solution of the Grad-Shafranov equation and the associated lines of constant ψ in the R, Z plane.

(b) An illustration of the resulting magnetic coordinate system as derived from the equilibrium magnetic field. (CCFE)

Figure 1.2: Illustrations of the equilibrium magnetic field in CENTORI and the resulting grid. Figures courtesy of Peter Knight (CCFE)[29]

The Grid

The coordinate system is a continuum whereas CENTORI operates on a structured finite mesh of points, requiring the coordinate system to be discretised into a grid. The resolution of the model defines how many points are placed in the ψ, θ and ζ directions. The magnetic axis does not form part of the grid as all values of $\nabla\theta$ are degenerate. Figure 1.2b shows a diagrammatic interpretation of the grid.

1.4.4 Integration

At any point in the simulation, CENTORI's primary quantities describe a snapshot of the state at a particular time in the simulation. CENTORI evolves the system state in time by solving a suite of differential equations which form an Initial Value Problem (IVP), which can be generally described by Equation 1.2.

$$\frac{\mathrm{d}\mathbf{y}}{\mathrm{d}t} = \mathbf{f}(t, \mathbf{y}(t)) \qquad \mathbf{y}(0) = \mathbf{y}_0 \tag{1.2}$$

There are many algorithms for solving Initial Value Problems, however one of the simplest is the explicit Euler step which requires the evaluation of $\mathbf{f}(t, \mathbf{y})$ once per time step of length, h.

$$\mathbf{y}_{n+1} = h\mathbf{f}(t, y_n) + \mathbf{y}_n \tag{1.3}$$

Truncation error

Solving IVPs numerically introduces cumulative truncation error at every time step, the difference between the true solution $\int_t^{t+h} \mathbf{f}(\mathbf{y}(t'))\mathrm{d}t'$, and the numerical approximation. The size of truncation error is related to the size of the time step, h, via an order relation, usually a power of h, the higher the order the "faster" the error decreases as h becomes smaller (it is presumed $0 < h < 1$).

The Euler method described by Equation 1.4 has a truncation error of order $\mathcal{O}(h^2)$, so the error scales linearly with the time step. Higher order methods, like Runge-Kutta methods, may have truncation error $\mathcal{O}(h^5)$, but require the evaluation of $\mathbf{f}(t, \mathbf{y})$ more than once per time step (generally n evaluations for an $\mathcal{O}(h^{n+1})$ method). It is therefore usually more efficient, to achieve the same accuracy, to use a larger time step and a higher order method than a lower order method with a smaller time step as it will require fewer evaluations of $\mathbf{f}(t, \mathbf{y}_n)$ to reach the same solution.

Explicit and Implicit

Equation 1.3 is an explicit solution, as the solution $\mathbf{y}_{n+1}$ is purely a function of the known state $\mathbf{y}_n$. This is relatively easily to solve as it only requires evaluating $\mathbf{f}(t, \mathbf{y}_n)$ with the data that is already known. An equally valid solution is Equation 1.4 where the solution, $\mathbf{y}_{n+1}$ appears on both sides of the equation.

$$\mathbf{y}_{n+1} = h\mathbf{f}(t, \mathbf{y}_{n+1}) + \mathbf{y}_n \tag{1.4}$$

This is an implicit formulation of the problem which does not affect the truncation but is more numerically stable, however usually requires the inversion of a large system of equations every time step which incurs a significant computational cost and may be difficult or impossible to derive for non-linear systems.

Time step lengths and stability

There are limits on the size of individual time steps in an explicit scheme described by the Courant-Friedrichs-Lewy (CFL) condition[30] for advection in Equation 1.5. It relates the time

Tokamak	B_0(T)	ρ_m ($\times 10^{-7}$kg/m^3)	V_A($\times 10^6$m/s)	Δx (cm)	h (ns)
RTP	2.4	1.0	6.8	1.3	< 0.2
JET	3.5	3.4	5.4	7.0	< 2
MAST	0.5	1.0	1.4	5.1	< 4
ITER	3.5	3.4	5.4	16	< 3

Table 1.1: Typical values of the Alfvèn speed V_A as calculated from characteristic features of current and future tokamaks and associated limits on time step (Assuming 128 grid points and $C = 0.1$). (ITER values estimated from scaling features). Table values for RTP, JET and ITER from [31], MAST from [32].

step, h, the maximum velocity in the system, u and the grid dimension Δx to a constant convergence criterion, C.

$$\frac{u.h}{\Delta x} < C \tag{1.5}$$

In the plasma simulations that CENTORI undertakes the fastest moving phenomena is the Alfvèn wave, which is defined by Equation 1.6 where ρ_m is the mass density and B_0 the strength of the magnetic field. Table 1.1 shows typical time step lengths of nanosecond order for simulations of various tokamaks.

$$V_A = \frac{B_0}{\sqrt{\rho_m \mu_0}} \tag{1.6}$$

Time step length affects the stability of the dynamical system, with excessive time step length in explicit systems causing solutions to diverge widely from their true solution. The negative feed back in implicit formulations tends to prevent numerical explosions caused by excessive time steps and are favoured for this improved numerical stability

1.4.5 Integrating CENTORI

CENTORI's solver uses a semi-implicit predictor-corrector scheme to evolve the primary quantities. The equations would most quickly and easily be solved using an explicit method, however there are expected to be some large derivatives during simulation and a more stable algorithm is preferable. It can also be shown that an explicit method will increase the total amount of energy in a dynamical system, whereas an implicit method will reduce it. To prevent non-physical diffusion in the model a semi-implicit method is used which is guaranteed under certain conditions to conserve the kinetic energy[33]. In the Euler step case this be formulated to:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + h\mathbf{f}\left(\frac{\mathbf{y}_{n+1} + \mathbf{y}_n}{2}\right)$$

Finding the semi-implicit solution directly is non-trivial as the core equations are non-linear which makes finding an inverse operation very difficult as it would also involve solving a large system of equations which would generate a large amount of computation. Instead CENTORI uses a Predictor-Corrector technique, where the solution is found iteratively. If the true analytical solution to the problem is:

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \int_t^{t+h} \mathbf{f}\left(\mathbf{y}\left(t'\right)\right) dt'$$

then CENTORI integrates $\int_t^{t+h} \mathbf{f}(\mathbf{y}(t'))dt'$ using a numerical quadrature, the midpoint rectangle method, where

$$\int_t^{t+h} \mathbf{f}(\mathbf{y}(t'))dt' \approx h\left(\mathbf{f}\left(\frac{\mathbf{y}_{n+1}+\mathbf{y}_n}{2}\right)\right)$$

which has a truncation error of $\mathcal{O}(h^3)$. However this method is still an implicit formulation of $\mathbf{y}_{n+1}$ and rather than performing a complicated inversion CENTORI creates a *Prediction* for $\mathbf{y}_{n+1}$ using the Euler method. This prediction is then used in the *Correction* iteration to produce a more accurate solution. Multiple iterations creates a sequence of $\mathbf{y}_{n+1}^m$ which should converge[1] to $\mathbf{y}_{n+1}$:

$$\mathbf{y}_{n+1}^{m+1} = \mathbf{y}_n + h\left(\mathbf{f}\left(\frac{\mathbf{y}_{n+1}^m+\mathbf{y}_n}{2}\right)\right) \tag{1.7}$$

CENTORI performs this evaluation a further three times as the error in the estimate is limited by the truncation error of the midpoint rectangle method, $\mathcal{O}(h^3)$.[34, 35]

1.4.6 Differentiation

Evaluating $\mathbf{f}(t, \mathbf{y}_n)$ requires performing differential operations like $\nabla \times \mathbf{A}$, $\nabla.\mathbf{A}$ and ∇A on vector fields. This requires a robust numerical differentiation scheme for which CENTORI uses a centred finite difference scheme with a truncation error $\mathcal{O}((\delta x)^2)$.

$$\frac{\partial y(n)}{dx} \approx \frac{y(n+1)-y(n-1)}{2\delta x}$$

As it is a finite difference scheme it is simple to implement, cheap to compute and resistant to shocks, though does cause numerical diffusion[35, 34]. Experiments have been conducted

[1] The radius of convergence has not been established and so the conditions which have to be satisfied to guarantee convergence have also not been established

with derivatives calculated from a Discrete Fourier Transform (DFT) which are theoretically capable of producing smooth derivatives. In practise, the DFT results were subject to ringing or "Gibbs Phenomena" when applied to the high gradients caused by CENTORI's boundary conditions. The finite difference scheme was retained because of the noise and the additional computational cost (3 to 12 times)[36].

Boundary conditions

Using the centred finite difference to evaluate $\frac{\partial}{\partial\theta}$ and $\frac{\partial}{\partial\zeta}$ is well defined for every point in CENTORI's plasma coordinates, as being angular coordinates and periodic, data exists for $y(n+1)$ and $y(n-1)$ for every point $y(n)$. This is not the case for $\frac{\partial}{\partial\psi}$ where points adjacent to the magnetic axis or the plasma's edge cannot be calculated.

Data for these points is defined by the boundary conditions of the problem. Typically, boundary conditions for the plasma's edge are a constant value like zero or require creating a condition like $\frac{\partial}{\partial\psi} = 0$ or $\frac{\partial^2}{\partial\psi^2} = 0$. Though no values are held for the magnetic axis point the value is interpolated from the points that surround it and used in the calculations.

Defining the correct boundary conditions for each quantity is an important part of the model and can have a significant impact on the stability of the simulation and accuracy of its results. The exact nature of the boundary conditions used for each quantity in CENTORI is beyond the scope of this explanation, though more detail on the performance of their implementation is presented later.

1.5 Optimising CENTORI

The previous sections define CENTORI's mathematical model and describe the algorithms and techniques that are used to evolve and integrate it forward in time. To run this on a computer, a far more detailed set of instructions have been written as Fortran 95 source which explicitly defines how the program will execute. The source code explicitly defines the exact operation of the simulation, from the way data is stored in the computer's memory to the order that calculations are performed.

Time on supercomputers is valuable and ideally computers and compilers would be sufficiently advanced that CENTORI would run optimally on every platform. Yet there are many ways to implement the operations and algorithms required by CENTORI and some will run faster than others. Combined with the fact that the computers CENTORI runs on are all

different then it is likely that CENTORI's performance could be improved on one or more of the architectures it runs on.

This project documents the efforts to optimise CENTORI's performance on the current generation of supercomputers and to provide a framework which will aid its optimisation on future generations.

Chapter 2

Modern High Performance Computing

The number of calculations required to evolve a simulation of tokamak plasma using CENTORI at scientifically meaningful temporal and spatial resolutions is enormous. Combined with the requirement to run multiple configurations of the model to establish statistical significance it could potentially take hundreds of years to complete the simulations on a "normal" workstation. Though the processing speed of individual microprocessors has consistently increased since their inception, a single processor still does not have the capacity to produce the required results within an acceptable timescale. If Moore's Law[37], the observation that processors performance doubles every 18 months, were to continue a 1024 fold increase in performance could be achieved in 15 years; however this is too long to wait!

Instead of relying on a single processor many individual processors are coupled together to calculate the solution in parallel. Solving problems in parallel dominates the design of most modern supercomputers (or High Performance Computers) with tens of thousands of processors interconnected across fast networks forming Massively Parallel Processors (MPP). The fastest of these computers offers performance that is hundreds of thousands times greater than a single processor could ever achieve but only on problems that can be parallelised. This chapter describes some of the characteristics of modern MPP systems such as HECToR[38], HPC-FF[39] and HPCx[40] that affect performance and optimisation.

2.1 The Processors

The power-house of every MPP supercomputer are the processors. The choice of processor in most modern High Performance Computers is a result of historical, social and economic trends in the wider Information Technology sector. Though there are modern descendants of the original supercomputer processors that are designed around scientific workloads, the continuing adoption of computing by society as a whole has created a new, cheaper, class of processors, the commodity processor[41].

A commodity processor is one that is typically applied to a wide variety of tasks, from embedded devices to database workloads, and as a result is sold in vast numbers. Such large production volumes allow economies of scale that reduce prices and increase the floating-point performance per dollar of each chip beyond that of dedicated scientific processors. These processors tend to be based on standard instructions sets, the most common being the x86(-64) instruction set[42] which are used used by processors from Intel and AMD and the Power RISC[43] instruction set in devices from IBM.

2.1.1 The Traditional Model of a Microprocessor

Nearly all modern code is written in high level "portable" languages like Java, C, C++ and Fortran, and HPC codes are no exception. Programming at the higher level is typically more productive than writing directly in assembly as it is less error prone and easier to understand and the results can be ported to more than one computer architecture. These high level languages operate on a general model of how processors behave which ensures the code can be run on multiple architectures. This is the traditional model, taught to students, that the processor is a very quick executor of simple instructions. The instructions can make the computer perform like a calculator, adding, multiplying and subtracting values, or as a filing clerk, shuffling information around a very large memory space. This model is a good approximation of the structure of the original computer architectures, however it increasingly masks the underlying complexity of modern microprocessor architectures.

The traditional model is a fetch-execute model which imagines the processor fetching the next instruction from a sequence in memory, decoding the instruction and performing an action based upon it. These instructions form general categories, some load or store from or to the main memory, some change the next instruction to act on (branch, procedures etc) and others perform operations on data. Typically all models assume that the computer has:

- An infinite memory address space, there is no limit on the amount of memory that can be addressed or that is physically available.

- A private memory address space, with each program being the only one using the computer and having sole access to a linear address space

- Uniform memory access, it takes the same time to access any point of memory in any order, the memory is Random Access Memory (RAM)

- Uniform instruction completion, it takes the same amount of time to complete every instruction, be that accessing memory or adding two integers.

Compilers translate the source code statements into native machine code whilst performing some level of optimisation on the resulting instruction stream wherever possible. In most cases the compiled code performs adequately and conventional wisdom is to not attempt hand optimisation of the code. This is adequate in many fields, like office software, database development or Internet applications, where the computer's time is far cheaper than a human's. However the scale of running on a supercomputer makes further human hand optimisation of applications economically viable.

2.1.2 Optimisation of Serial Code

Modern microprocessors represent some of the most complicated machines ever devised by humanity. A typical microprocessor is manufactured from millions of individual transistors all working at high speed with sizes smaller than the wavelength of visible light; in 2009 a quad core "Barcelona" AMD Opteron processor contained approximately 758 million transistors over an area of only 285mm^2[44] and a quad core "Nehalem" Intel i7 processor contained 731 Million transistors[45] in 263mm^2. Compare this to a Boeing 747-400 which is constructed from approximately six million individual components[46] yet costs many millions times more than a individual microprocessor which is available to the general public in any electronics store.

This complexity is required to support the processor's ever increasing capability to process data and perform calculations, both on integer and floating-point numbers. The performance has improved year on year in line with the self-fulfilling prophecy that is Moore's Law, but while the processor is the central focus of application performance it is ultimately dependent upon many of the other subsystems which form a "computer". Some of the features which are used

to improve microprocessor performance in past, present and (probable) future MPP systems are described in the following sections.

While compilers are written with a deep understanding of this complex hardware, they are unable to understand the code that they are being asked to process in the same way as a human being. This means it is possible for hand optimisations of the code by a human, that rely on a better understanding of the strengths and weaknesses of the underlying hardware, to achieve better performance than a compiler application.

2.2 Floating-Point Performance

Most scientific applications model real world situations and so store data as floating-point values. Most processors can theoretically perform many billions of floating-point operations per second (FLOPS), however they can only do this under specialised circumstances. By further understanding how processors perform floating-point operations, including the many levels of instruction level parallelism within the processors, it is possible to arrange operations at the highest level so that the code can be executed faster by the processor.

2.2.1 Pipelining

The fetch-execute model assumes that the processor completes every instruction before moving on to the next one in the sequence. However, executing an instruction on a modern processor can be broken down into several stages. An example sequence for a floating-point instruction is:

1. Fetch the instruction from memory (or cache; see Section 2.3.1).
2. Decode the meaning of the instruction, which can be difficult (i.e. time consuming) on Complex Instruction Set Computer (CISC) like x86(-64), or simple (i.e. quick) on Reduced Instruction Set Computer (RISC) architectures like Power.
3. Floating-Point unit stage 1, perform the first half of the floating-point operation on the operands
4. Floating-Point unit stage 2, perform the second half of the floating-point operation.
5. Write the result back to the register file, a short term scratch memory space for storing operands and results.

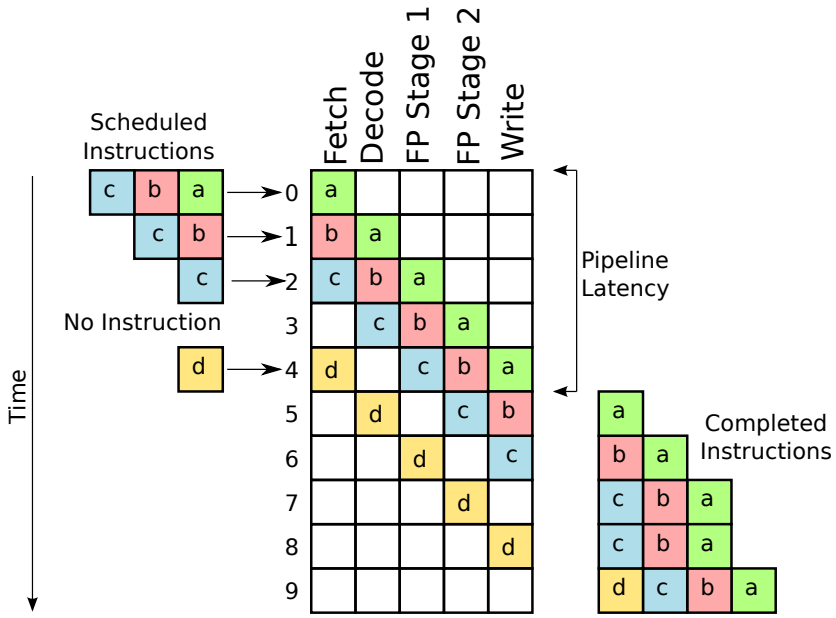


Figure 2.1: An example five stage pipeline for floating-point instructions. A pipeline stall between the 3rd and 4th instructions issued is shown and the pipeline latency of 5 instructions is shown before an instruction completes every cycle.

Each of these operations is assumed to take one clock cycle to complete, so an individual instruction would take five cycles to produce a result. However once the instruction has passed the fetch stage a new instruction can be fetched while it is being decoded. Rather than leaving individual functional units idle in a five stage process like the example above and Figure 2.1, five instructions can be processed simultaneously, keeping all the hardware in full use at every clock cycle. This increases the overall throughput of instructions from one every five clock cycles to one every cycle.

This is a form of parallelism that effectively increases the rate at which operations are processed by the number of stages in the pipeline assuming each stage takes an equal amount of time. There are many different pipelines in a computer architecture, and this example shows how the individual arithmetic logic units are themselves pipelined in order to increase the rate at which operations are completed. However this can only be achieved when there are a sufficient number of operations to be performed. Instructions can only be pipelined if they are not dependent upon the results of instructions still being processed and if there is data available. Keeping the pipelines full of instructions is an important part of achieving good performance on most processors[47, 48, 49].

In the case of floating-point instructions, on most processors only some are pipelined. Typically, floating-point instructions like divide and square root are not pipelined meaning only one

instruction can be processed at once which can take many 10s of cycles to complete. This massive disparity between the rate at which additions, subtractions and multiplications complete and the rate of divides and square roots means there is a significant performance advantage, exploited by optimisers, in removing these operations or converting them to some other form (e.g. multiplication by the inverse in place of a division).

2.2.2 Speculative Execution

Achieving a consistent flow of data and instructions to the pipeline is important to the overall performance of the processor. Any time data or instructions are unavailable the functional units are left idle and the instruction throughput is reduced. This becomes most important when the program's execution pathway is dependent upon the result of an operation. Many codes have branch (`if`) statements that change the execution pathway depending upon the result of a calculation (e.g. `if ((a+b)>0)`). In such a case the execution of the next sequence of instructions will be delayed until the result of the operation can work its way through the pipeline and complete. This leads to stalls in the pipeline which reduce the overall performance of the code.

To limit these stalls hardware designers allow the processor to speculatively execute instructions, where it will continue to process assuming the result of a branch operation based upon a prediction. The branch prediction takes into account how many times a particular path has been followed and chooses the most common. If the prediction was correct the processor carries on uninterrupted, if it was wrong the processor must roll back to the correct point and continue down the correct path. This method of prediction is successful for loops where the code will follow the same path in each iteration until the last case. This technique keeps the queue of instructions full and prevents stalls in the processor pipeline improving the overall performance.[48, 49]

2.2.3 Super-scalar and out-of-order execution

Processors have different hardware for performing different types of operations, and most are capable of performing different instructions on each of these functional units, executing different types of instruction simultaneously. For example, it is possible on most architectures for integer and floating-point instructions to be processed simultaneously, increasing the overall throughput of the processor.

The ability to do this is highly dependent upon the parallelism of the instruction stream, which is usually dependent upon the properties of the compiler used. The processor has to analyse the dependencies of each instruction so the compiler has to generate an instruction stream which can be parallelised, removing any unnecessary conflicts caused by reusing the same memory locations. However in some cases the next instruction in the sequence may have dependencies which prevent it from being dispatched through the processor straight way. In these cases the processor will read ahead in the instruction stream to find an instruction it can execute. Many modern processors are capable of executing up to four separate instructions at any one time though they must keep track of the order of instructions and make sure the evaluation is correct[48, 49].

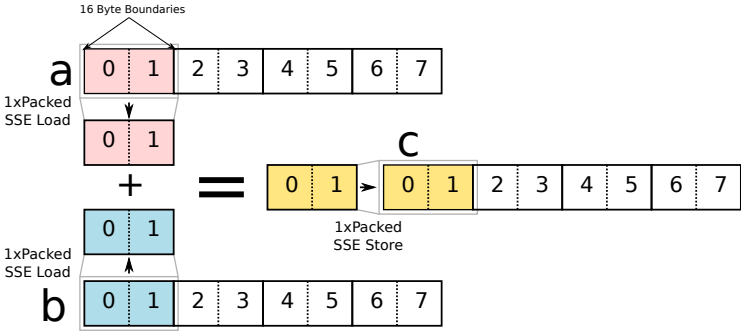
2.2.4 Simultaneous Multi Threading (SMT)

Ultimately despite the efforts of the processor there are circumstances where there are no further instructions to be executed in parallel by the hardware. In these cases the processor can either wait (stall), or attempt to execute another, completely different set of instructions that do not depend upon the results of the delayed instruction. The processor can support multiple simultaneous threads of execution which it can switch between when one stalls and continue processing. Most general purpose processors with SMT present a single processor as two logical processors which can operate simultaneously though it is possible to present more. The approach is extended in specialist barrel processors which can have hundreds of threads operating concurrently or on Graphics Processing Units[50]. Optimising the performance with these architectures depends upon application level parallelism which usually has to be identified by the programmer[51, 52].

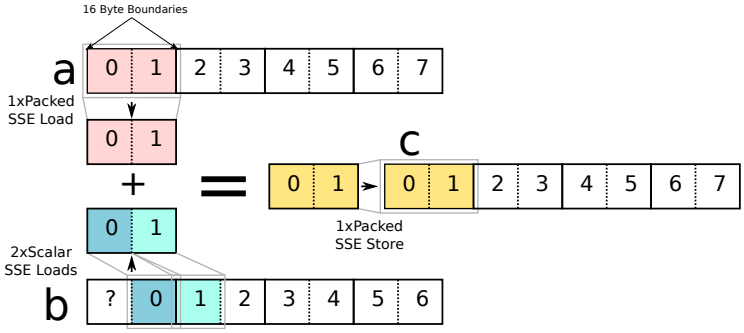
2.2.5 Vectorisation

Ultimately more operations can be completed per clock cycle by adding more floating-point hardware and allowing more than one operation to be performed in a single instruction. Instruction sets either natively support, or are extended to include, special instructions which can perform the same operation on vectors of data. This approach is categorised as Single Instruction Multiple Data (SIMD) and has multiple advantages for processor performance.

As well as performing more operations simultaneously it reduces the number of instructions required to perform the same operation using scalar instructions. This reduces the memory



(a) An example SIMD operations on aligned data in all three arrays.



(b) An example SIMD operation on data aligned in two arrays but unaligned in the third.

Figure 2.2: SIMD SSE operations to perform the array operation `c(:)=a(:)+b(:)` with completely aligned arrays (Figure 2.2a)and one misaligned array (Figure 2.2b).

bandwidth required to fetch, decode and execute instructions in the processor which allows more bandwidth for fetching and storing data. The parallel nature of these instructions places stringent requirements on the alignment of the operands in memory (usually operands have to be aligned on 16 byte boundaries), otherwise multiple load and store instructions have to be performed which erodes some of the advantage of SIMD instructions.

Figure 2.2 shows how the array operation `c(:)=a(:)+b(:)` is being performed using SIMD instructions capable of pairing operations together. In Figure 2.2a the registers can be loaded with a single "packed" instruction as all the data is aligned on the 16 bytes boundaries. In Figure 2.2b the data in `b(:)` is misaligned compared to `a(:)` and `c(:)` and so requires two scalar half load operations to pack the register, reducing the overall efficiency of the operation.

The x86(-64) instruction set was augmented by the Streaming SIMD Extensions instructions (SSEs) which includes support for SIMD instructions on two double-precision floating-point values and four single precision floating-point values. Future extensions to the x86-64 instruction set will include support for even wider vectors, like Intel's AVX[53]. Earlier generations of x86-64 processors did not have parallel floating-point units so would serialise a SIMD instruction

into multiple scalar instructions. Modern AMD and Intel processors have extended the vector floating-point pipeline to 16 bytes and so can now process both parts of a packed SIMD operation simultaneously, doubling the maximum theoretical floating-point performance.

Power processors have the optional AltiVec instruction set extension which perform a similar role, though is not currently available on any of the target architectures. The BlueGene/L's processor has a DFPU unit which is also capable of performing two floating-point operations in a single cycle.

2.3 Memory Performance

Though capable of processing many billions of operations every second a processor has a very small short term memory that it can access instantly. This short term memory, the register file, holds the inputs and results of calculations and processor book keeping details like the location of the next instruction to execute. Most applications require a much greater storage capacity which is held on the much larger and long term (though volatile) main memory system. In the standard model of the computer, the main memory is Random Access Memory (RAM) where it costs the same to read or write any location in the memory address space. The data is transferred into and out of the processor registers via instructions issued by the processor.

It follows that the rate of operations that a processor can complete is directly related to the rate data can be transferred to and from the processor. If all of the operations required only need 10 to 100 operands of local storage, the data could be stored in the register file, however every real code will have to make use of main memory so the processor is forced to operate at the maximum rate that data can be streamed from the main memory, memory bandwidth. For scientific codes like CENTORI, which operate on large amounts of data one of the most important considerations is optimal use of the memory subsystem. The rate at which data can be streamed to and from the external memory is often the fundamental limit on the performance applications so they have to be designed to optimise the bandwidth available to them. Increasing this bandwidth, or performing more floating-point operations per byte transferred will increase the overall performance of the code on the processor.

The delay, or latency, between issuing a read instruction and the data arriving in the register can be many hundreds of processor cycles because main memory operates at a much lower frequency. This latency is the result of physical limitations, the memory operates at a lower clock speed and is a separate module in the system. In 2009 most processors operate at frequencies

of GHz whilst main memory operates at 100s of MHz. Typically a processor can wait idle for 700 cycles for an outstanding data request to main memory potentially performing no useful work. As the gap between the performance of main memory and the processor has widened hardware designers have attempted to bridge it. Almost every modern processor has additional circuitry dedicated to reducing the effect of long latencies and low bandwidth in the main memory subsystem and to increase the bandwidth. An awareness of these techniques allows application programmers to optimise their code on these architectures.

2.3.1 Caches

The primary feature used to address memory latency issues are the caches of data (and instructions) that are held in pockets of faster memory close to the processor. Caches are separate sections of memory, usually contained on the same chip as the processor, which have a much lower latency and higher bandwidth than the main memory modules. The caches are typically much smaller than the main memory, but much larger than the register file, usually capable of storing kilobytes of data.

Most codes that reload data from main memory usually do so only a short time later (temporal locality), and usually access data in nearby address locations (spatial locality). By storing the most recently used values and their nearest neighbours in fast caches, if the data is required again or its nearest neighbours it can be restored much more quickly than a request to main memory. Common MPP processors have a Level 1 cache that is split between data and instructions each of which varies between 32KB - 64KB in size and takes only 10 cycles (approximately) to access data in them[54]. Data is stored in contiguous groups aligned on memory boundaries (usually 64 or 128 bytes), to form cache lines or blocks, so subsequent accesses to memory addresses nearby will already be cached.

Associativity

In order for a cache to function it must be able to quickly establish whether a memory addresses exists as a copy in the cache. In a fully associative cache, where copies can be held in any part of the cache, each location must be checked on every memory access to find a match. This has the advantage that data is only removed from the cache when every other location has been used, but is expensive to implement on the chip and time consuming to perform.

The simplest alternative is to assign a single location in the cache for each memory address

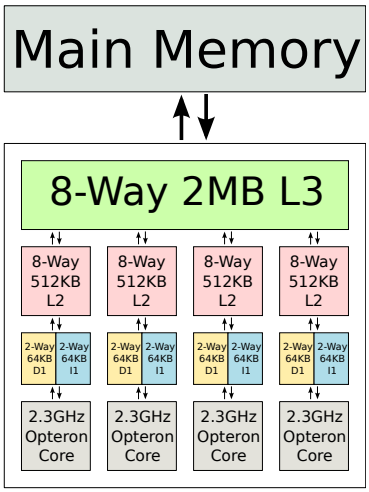


Figure 2.3: The memory hierarchy of a quad core Barcelona Opteron processor. Each processor core has a unique L1 and L2 cache and a shared L3 cache. Data is loaded directly from main memory to the L1 cache, whereas data is only written back to the main memory when data is finally evicted from the L3 cache.[55]

but because the cache is much smaller than the main memory there will be many memory addresses mapped to cache location. It would now be much simpler to check if an address was already in the cache but each time another address was accessed that mapped to the same site the existing data would be evicted, even if some there are some unused parts of the cache.

A compromise is an N-way associative cache, where each memory address is assigned to one of N locations in the cache. This prevents data from being evicted unnecessarily and reduces the potential number of locations to be searched. Common levels of associativity are 2-way, 4-way and 8-way, with 8-way being almost as efficient as a fully associative cache under test[56].

Cache Blocks

The minimum amount of data that can be copied from main memory into the cache is a “block” or “line”. This is a contiguous collection of addresses, usually of the order of 32 to 64 bytes, which is loaded into the cache whenever an address within the block is accessed. Loading data in larger blocks improves the latency of the cache (fewer locations to search) and is a way of prefetching spatially local data. However, loading all the data in a cache line causes more data to be loaded from main memory than is absolutely necessary if all the data in a line is not required. Accessing data in strides greater than a cache line is an example of this problem.

Eviction and Replacement Policy

Caches inevitably become full and existing data has to be evicted to make room for the incoming data. In a write-through cache, the data is written back to main memory as soon as it is stored in the cache generating large amounts of memory traffic, whereas a write-back cache waits until eviction to write data back to the main memory. Ideally, eviction would only occur to data that will not be used in the near future, however this is very difficult to predict. As most caches are N-way associative they have a choice of N items to evict. Choosing at random is one replacement policy, however usually a cache will use a Least Recently Used (LRU) algorithm (or a cheaper approximation) which achieves better performance by evicting the "oldest" piece of data first. On caches with low associativity (i.e. 2-way) there is such a limited number of candidates for eviction that data can often be evicted only to be reloaded again a short time later. This can cause cache thrashing if arrays are used that have lengths which are multiples of the cache size as each will be competing for the same locations in cache.

Hierarchy of Caches

Rather than building a bigger single cache, which would be slower, more expensive and consume more power, most systems incorporate multiple levels of cache[57]. As the cache moves further from the processor it grows in size but the access time increases. Typical commodity processors have three levels; small, fast L1 caches, medium speed and size L2 caches and larger, slower and potentially shared L3 caches.

These caches operate on the same principle as the top level cache, but are able to store more data for longer due to their size. There are different policies for which data should be stored in each cache, though a common method is to store the evicted values from the higher levels of cache on the next level as a "Victim Cache"[58]. Other systems will keep multiple copies of data in each level of the cache.

Cache Misses

When a program attempts to read from a memory location the caches are checked to see if a copy exists. If it does not it has to be fetched from main memory which may take many hundreds of cycles. However, when a program attempts to write to a memory location, it may be expected that a copy does not already have to exist in a cache. However because caches operate on lines and blocks of data, the process must read the entire line into the cache so it

can alter the single location and write the resulting line back to main memory. Thus a write miss on the cache will incur at least as much (if not more) of a penalty than a read miss, though some processors do include instructions which can prevent the load if all the data in a cache line is to be written.[49]

2.3.2 Pre-fetching

Caches reduce the latency of data that has already been accessed, but in many cases data has to be fetched from main memory for the first time, causing a compulsory miss. To reduce the impact of memory latency data can be pre-fetched in advance so it is ready to use when needed. Modern architectures have multiple methods of doing this; hardware pre-fetching monitors the access patterns of the instruction stream and attempts to predict what data will be required and loads the data directly into the cache before it is required. This can be very effective with uniform linear memory access patterns which most loops and codes will use and most memory subsystems include hardware to do this[59]. This can be defeated by random or complicated access patterns which defy prediction, or overloaded if there are too many streams requesting different sequences of memory.

Alternatively, most instructions sets provide instructions which can be issued which will load a memory address directly into the highest level cache. These have to be included by the compiler[60] (or programmer) and offer a hint (the architecture is usually free to ignore the request) to the memory system of future data requirements[58, 61]. Inclusion of these is a fine balance, as requests have to be timed early enough to allow the data to be transferred, but not so early as to evict useful data or be evicted themselves before being used. This balance is especially fine when operating on low-associativity caches as eviction of still relevant data is much more likely

2.4 Interconnecting Processors

To run an application in parallel requires that the individual processors can communicate information between one another. Just as in memory access, there are two important properties of any link between processors:

Latency

Latency is the delay between the sender initiating a communication and the time before the data begins to reach the receiver. Latency is often a function of the physical properties of the systems like the length of the cables separating nodes. This means latency is different between different nodes in the system, with closer nodes having a lower latency than more distant pairs. As it takes 1 nano second for light to travel 30cm the physical layout of some machines which approaches 10s to 100s of metres can have a significant impact on the latency.

Bandwidth

Bandwidth is the rate at which data is transferred between processors, it is an important factor in large transfers, where the time taken to transfer the data is more significant than the latency. Bandwidth is usually a function of economic rather than physical limitations as adding more links between the sender and receiver will increase the bandwidth between them.

2.4.1 Types of interconnect

There are two major types of interconnect between processors:

Shared Memory

Multiple processors can be connected to the same main memory subsystem through a shared address space. Each processor can read and write to every address so processors transmit information by writing to an address in memory then having other processors read from it. This is a relatively simple programming model that can be applied to many loops and recursions in applications that are inherently parallel. Each processor can operate on a different iteration of the loop. This approach, however, does not scale beyond a few processors because while there is only very low latency in a communication, as processors are added there is increased contention for the memory subsystem. The bandwidth of the communication is the effective memory bandwidth of the system which has to be shared between the processors ultimately reducing the rate at which each operates.

Networks

Multiple processors are connected over a network, each processor can transmit data to every other processor by sending and receiving messages. This approach allows each processor exclusive access to its own memory and so has peak bandwidth. This approach scales more readily as more links can be set up across the network to increase the bandwidth and reduce the contention allowing hundreds of thousands of processors to be connected. Latency tends to be quite high as processors tend to be separated by multiple layers of hardware (network interface controllers, routers etc) as well as being physically separated. Programming these systems is more complicated as messages must be actively transmitted and received by each processor requiring a fundamental restructuring of most serial programs.

Most modern systems are a hybrid of the two techniques using a network to connect many shared memory systems. This has the advantage of allowing the low latency communication between processors on the same shared node (intranode), whilst also allowing the large scaling potential of a message passing network. Messages can be passed between nodes with shared memory without using the network. Shared memory messages typically have lower latency and higher bandwidth than messages passed over the network. Traditionally most applications have only used the message passing model on hybrid systems, however it is possible to mix shared memory and message passing which can be advantageous when the number of processors sharing a single network port is large.

This approach has been reinforced by the trend of vendors of modern commodity processors to include many identical cores in a single package. These multi-core processors act like a shared memory architecture and form a hybrid architecture when networked together. This trend is likely to continue as the number of cores per processor packages continues to increase and as heterogeneous architectures like mixed Processor + Accelerator models are offered by vendors.

2.4.2 Network Topology

Nodes can be connected together on the network in many different ways. The greater the number of links between nodes the higher the bandwidth between them (providing the network is capable of routing data down multiple paths). The greater the number of network links between nodes the greater the latency of the communication. If every node on the network has to communicate with every other node then each will only have access to a limited proportion

of the overall bandwidth of the network. Each of these factors is determined by the topology of the network. There are three common types of topology — toroidal mesh networks, trees (including fat trees) and hypercubes[62].

2.5 Message Passing Communications

To scale beyond a small number of cores applications need to pass information over the network. To maintain consistency across platforms, rather than using a mixed mode approach, most codes use message-passing internally on shared memory processors. Typical types of communication required by a message passing code are:

Point-to-point communication

Point-to-point communications pass information between pairs of processors. The transmitting processor issues a "send" command, transmitting data to a target processor which must be matched by a corresponding "receive" command on the target. Send commands define a buffer of data on the source which will be copied over the network into another buffer defined by the receive command on the target. In many cases each processor will block execution of the program and wait until the send or receive operation is complete and all data has been sent or received forcing every send to wait for the corresponding receive and vice versa.

Alternatively processors may use an asynchronous mode of operation that allows sends and receives to be performed in the background while computation continues. Received data cannot be accessed and the sent data destroyed until the transfer has been confirmed complete by the library. If there is hardware support computation can continue while the communication occurs concurrently in the background. In the ideal case processors would never be required to wait for data to be sent or received and would be constantly performing useful computation while communication occurred in the background.

Collective Operations

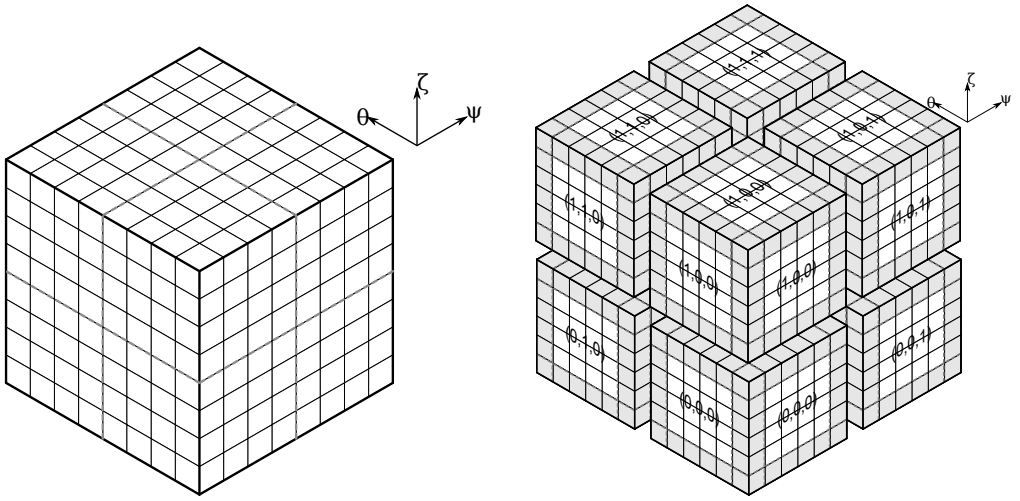
Collective operations involve groups of processors. Reductions perform an operation on data presented by a group of processors, this may be a summation or finding the maximum or minimum value (or any other user-defined operation). These operations could be performed with a sequence of individual send and receive operations, however specific operations allow the implementation to use the most efficient algorithm.

These operations almost always block the program until they are complete and in most cases, because of the data dependency, require that all members of the group enter the operation before the first may exit. These reduction operations form an implicit barrier to progress which causes all processors to become loosely synchronised after the operation is complete. Global operations involving small amounts of data like reductions and broadcasts are susceptible to communication latency and without special hardware they tend to be evaluated using a tree algorithm, each level of which will be subject to the cumulative latency of the link.

2.6 Writing Parallel Applications

Many scientific applications require a faster rate of computation than a single processor can offer, instead they are run in parallel on the many individual processors of MPP machines. The amount of parallelism within an individual application depends upon the problem being solved, some applications can be split into largely independent tasks that are only loosely coupled and are well suited to running on loosely coupled computers. Volunteer computing efforts like Seti@Home[63], Folding@Home[64] or climateprediction.net[65] are very successfully exploiting highly distributed computing resources connected over the Internet. Such tasks are colloquially described as "embarrassingly parallel" and require so little communication that it is usually inefficient to run them on a massively parallel supercomputer with a bespoke high bandwidth, low-latency network. These are best used for tasks that require a tight coupling between threads by frequently sharing information between the processors.

The first parallel supercomputers could split problems over only a few processors (<10), however as commodity processors have themselves become collections of multiple processing cores and frequencies have been limited by power consumption and heat dissipation the total number of parallel processing cores has increased rapidly. In 2010 most supercomputer class computers have tens of thousands of processing cores and the largest, petaflop scale, systems are created from hundreds of thousands of independent cores[66, 67]. The reliance on parallelism to deliver the current and future performance requires that CENTORI be capable of running in parallel.



(a) The dotted lines mark the boundary between processors when decomposing a $8 \times 8 \times 8$ field between $2 \times 2 \times 2$ processors.

(b) Additional halo data is stored on each processor (grey) which replicates the data from the nearest neighbour.

Figure 2.4: Illustrations of the decomposition of a 3D field in CENTORI and the resulting halo data.

2.6.1 Splitting CENTORI between processors

CENTORI is a data parallel application, following the Single Instruction Multiple Data (SIMD) paradigm[68]. It divides the global data structures equally between each of the processors in the calculation, with each processor performing the same actions on its private data. Global data in CENTORI can be visualised as a collection of large three dimensional arrays, which are split into smaller segments, each the same size, forming a virtual Cartesian grid of processors and data. Each processor is assigned a three number coordinate that describes the position of that processor in the global grid which allows it to determine its nearest neighbours.

To perform a finite difference calculation, or any other operation which depends upon data that is not held locally on the processor, it must be able to access the data held in other processors. Copies of the nearest neighbour data from the surrounding processors are held on each processor for this very eventuality which must be kept up to date as the change on the owning processor if they cannot be calculated locally. Local arrays are slightly extended to accommodate these "halos" of duplicated data as illustrated in Figure 2.4b.

2.6.2 CENTORI's communication Patterns

To communicate between processors CENTORI uses the Message Passing Interface (MPI), a standard Application Programming Interface (API) which is implemented by supercomputer

vendors and allows code to be written once and run at multiple sites. MPI provides calls for sending and receiving data between individual processors (e.g. sending halo data between processors) and collective operations that involve groups of processors. CENTORI requires communication between processors for three different operations:

Halo Exchanges

Halo exchanges are required because each of the global fields, like ion velocity or the magnetic field, are distributed between processors and the finite difference algorithms used require the nearest neighbour elements. In most cases the halo values are calculated along with the main field, however when a finite difference operation is performed the data must be refreshed from the neighbouring processors. This requires a sequence of point-to-point send and receive operations between each processor and its nearest neighbours, as halo data is exchanged between them. The amount of data exchanged increases with the number of processors as the halo volumes are directly related to the total surface areas of the local processor data volumes.

Halo exchanges send data between nearest neighbour processors first in the x processor direction (equivalent to the ψ direction), then the y processor direction (equivalent to the θ direction) and finally the z direction (equivalent to the ζ direction). By increasing the halo at each stage to include points which are part of more than one halo (points on diagonals), data is transferred from the nearest neighbours on the diagonal without specifically communicating with them. Boundary conditions in the θ and ζ directions are dealt with in the same way as nearest neighbour halo exchanges are, and collective reduction operations are used to sum values over the θ and ζ directions in order to evaluate flux surface averages.

Boundary Conditions

CENTORI's coordinate system is periodic in the two angular coordinates which means that all processors have a θ and ζ nearest neighbour, however in the radial ψ direction there are edges. One at the edge of the torus and the other at the degenerate central coordinate point. Scientifically formulated boundary conditions are defined at these points, at the torus edge these are functions of the data already held by each processor and require no further communication, however for the central point an average value of all the surrounding points is used. This requires a reduction operation to sum all the data held on the processors adjacent to the central point. As the number of processors in the θ and ζ directions increases the number of processors involved

in the communication increases and the time spent performing the reduction increases.

Integrations

Due to the physical properties of the tokamak, certain fundamental plasma quantities are measured on the flux surfaces which are surfaces of constant ψ. Integrating across these flux surfaces requires an `MPI_Scan` operation which performs a partial sum across processors in the ψ dimension, allowing the integral to be calculated. This operation is a form of reduction and due to the nature of the operation introduces a stagger, with processors at the beginning of the sum being released much sooner than the final processor which must wait for every other processor. As the number of processors in the ψ direction increases, more members are included in the `MPI_Scan` operation and the cost of the operation increases.

2.7 Measuring Parallelism

To optimise CENTORI's parallel performance it is important to understand how to measure parallelism and some of the metrics. Probably two of the most important concepts are scaling and speed up, which is the most intuitive. Ideally a completely parallel application which takes time T_1 to complete on 1 processor will take $T_N = \frac{T_1}{N}$ when applied to N processors. This is perfect scaling which is unlimited and given any execution time ϵ there exists a number of parallel processors, N_ϵ, which the application could run on such that the application would complete in time ϵ. The overall speed up of the application is measured by calculating $\frac{T_1}{T_N}$, which in the perfect scaling case is N.

Just because CENTORI has been written to work in parallel does not instantly cause CENTORI to run N times faster when applied to N processors. CENTORI's parallelisation is not a perfect distribution of work, not all of the computational work is executed in parallel, some must progress in serial like the initialisation of the model that solves the equilibrium force balance (Grad-Shafranov) equation[21], though this is only done once for each run configuration. The time spent copying the data between processors actually grows as the number of processors increases and other communications increase as the number of processors rises.

There are many synchronisation points in any parallel code. These are the points where processors are relying on other processors to send or be ready to receive data before they can continue and must wait idly by until the other processors catch up. Any grouped communication introduces a synchronisation point as all processors must wait for all the data to become

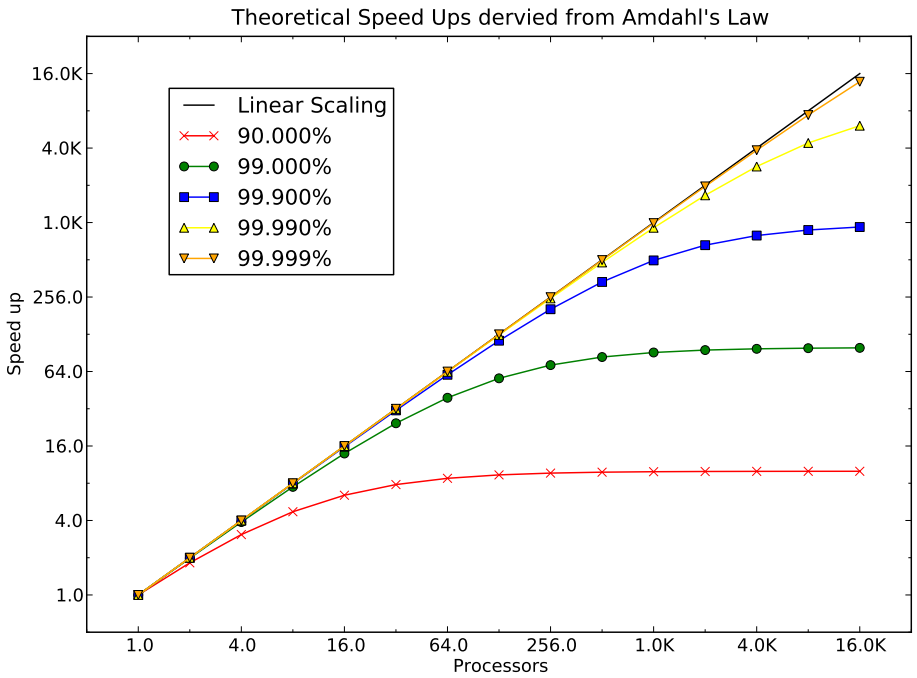


Figure 2.5: The effective speed up of an application for a variety of parallel fractions, P, on N processors modelled using Amdahl's Law.

available before the operation can complete, which implies waiting for the slowest processor to catch up. Explicit synchronisation points in the form of barrier operations, which wait for all processors to reach a set point before allowing any to continue are available in MPI, however they are not required in CENTORI and would cause unnecessary synchronisation. Halo exchanges cause a synchronisation point as each processor must wait for each of its neighbours to send the halos before it can continue and for processors to accept the halo data it sends, as all processors exchange data between their neighbours all processors will be loosely synchronised when exiting a halo exchange point. All of these factors contribute to the scaling of the code, an important metric in the analysis of a parallel application.

2.7.1 Amdahl's Law

In the real world parallel applications, like CENTORI, will have imperfect parallelisations meaning there is a proportion of the application that executes in serial (or at least not on all the processors). The relationship between the parallel and serial components of a program is modelled by Amdahl's Law[69] (Equation 2.1), where S represents the proportion of the

simulation that executes in serial, $0 \leq S \leq 1$, and P the remaining fraction that can be executed in parallel.

$$T_N = T_1(S + \frac{P}{N}) \tag{2.1}$$

This relationship demonstrates that there is a limit on the scaling when $S \neq 0$ which limits the speed up in a real application. Figure 2.5 demonstrates the speed up that may be achieved in a hypothetical application with different values of P and S using Amdahl's law. This example demonstrates that to achieve a speed up of around 10^4 from 10^4 processors requires the application to be 99.999% parallel.

Amdahl's law is a limited model as it assumes that problem sizes remain fixed as the number of processors is applied. This perspective is referred to as "strong" scaling (more processors are applied to solve the same static problem). In reality most users will increase the size of the problem being solved as the number of available parallel cores increases to introduce more parallelism. This is "weak" scaling and is widely used in scientific domains where increasing the resolution of a grid or including more particles can improve the accuracy or scientific validity of a result. However, this approach cannot be readily applied to CENTORI as the resolutions $128 \times 128 \times 128$ tested are approaching the limits of the fluid approximation used by the physical model and increasing the resolution of the simulation is of limited scientifically benefit.

Chapter 3

Investigating Serial Microprocessor Performance for CENTORI

3.1 HPC Architectures of Interest

Though CENTORI can expect to run on multiple generations of hardware over its lifetime CENTORI's performance can only be measured on systems that currently exist. There are three systems that are directly relevant to CENTORI: HPCx, an EPSRC funded service for the UK; HECToR a successor system to HPCx also funded by EPSRC and other research councils and HPC-FF, an EFDA funded system dedicated to fusion research. These systems are typical of modern high performance computing architectures and are all based on the commodity processor model with many shared memory nodes. Each node is constructed from processors with many cores and has a large shared memory. Each node is connected using a high bandwidth, low latency network which in the case of HPCx and HECToR is a proprietary vendor product while HPC-FF has a "commodity" Infiniband based network. Table 3.1 shows some of the core statistics of each architecture from material published by their manufacturers. Comparing these figures allows the systems to be compared to a first approximation, however the real world performance of CENTORI is the product of many complex interactions between the individual components.

	HPCx	**HECToR**	**HPC-FF**
Vendor	IBM	Cray	Bull
System Type	eServer 575	XT5h	NovaScale R422-E2
Processor Architecture	Power 5+	AMD Opteron	Intel Xeon X5570
Cores Per Node	16	4	8
Number Nodes	160	5564	1080
Total Processors	2560	22 656	8 640
Memory Per Node	16 GB	8 GB	24 GB
Memory Per Processor	1GB	2 GB	3 GB
Memory Type	Unavailable	DDR2 800	DDR3 1066
Memory Peak	Unavailable	6.4GB/s	8.5GB/s
Processor Frequency	1.5 GHz	2.3 GHz	2.93 GHz
Processor FLOPs	6 GFLOPs	9.2 GFLOPs	11.7 GFLOPs
Total FLOPs	15.4 TFLOPs	208 TFLOPs	101 TFLOPs
Interconnect Type	High Performance Switch	Sea Star	Infiniband
Interconnect Topology	Tree	3D Mesh	Fat-tree

Table 3.1: Theoretical performance statistics for HPCx, HECToR and HPC-FF.

	IBM Power 5	**AMD Opteron**	**Intel Xeon**
System	HPCx	HECToR	HPC-FF
Clock Speed	1.5GHz	2.3GHz	2.93GHz
L1 Data Cache Size	32KB	64 KB	32KB
L1 Set Associativity	4-Way	2-Way	8-Way
L2 Cache Size	2MB	512KB	256KB
L2 Set Associativity	10-Way	8-Way	8-Way
L3 Cache Size	128 MB	6 MB	8MB
L3 Cache Shared Between	8 Processors	4 Processors	4 Processors
Symmetric Multi Threading	2 Way	None	2 Way
Floating-Point Units	2 Fused	1 SSE Add	1 SSE Add
	Multiply Add	1 SSE Multiply	1 SSE Multiply

Table 3.2: Processor features of the processors used in HPCx, HECToR and HPC-FF.

3.1.1 Processor Performance

Each of the systems uses a different commodity processor from a different vendor, IBM, AMD and Intel, and each processor theoretically offers a similar order of floating-point operations per second (FLOPs). Achieving this peak figure is dependant upon many stringent and unrealistic conditions so a traditional way to compare the performance of a processor and memory subsystem architecture is the STREAMS[70] benchmark. STREAMS runs a series of simple kernels on varying sized arrays of double precision floating-point data which stress both the floating-point units and the memory subsystem in a similar way to an application like CENTORI. By measuring the rate at which the operations complete and the resulting flow of data into and out of the processor the bottleneck in the system can be identified.

Rather than running the four originals tests in streams two kernels inspired by the STREAMS benchmark were chosen: a simple addition of two arrays writing out to a third

$$\mathbf{a} = \mathbf{b} + \mathbf{c} \tag{3.1}$$

and the "triad" kernel, which scales the second operand by a scalar constant first

$$\mathbf{a} = \mathbf{b} + k\mathbf{c} \tag{3.2}$$

These are very simple kernels which it is easy for the compiler to process and optimise and represent a large class of operations performed by CENTORI. As they are simple, it is quite likely they will get the best performance from the hardware. Each simulation was performed on only one processor of each "node" with uncontested access to the memory subsystem (in a real run of CENTORI it is likely that all of the cores would be active and contending for access to memory).

Figure 3.1 shows the results of the benchmark for two of the three processor architectures, HPC-FF and HECToR. The detailed performance characteristics shown in the graphs are discussed in more detail later in the chapter, however some features to note are that the performance of all the processors decreases markedly when operating on large arrays as a result of the memory architecture of each processor. A full wider discussion of memory architecture and caches is found in Section 2.3.1, though it is possible to observe a stepping effect as the data no longer fits in the fast caches and is forced into a lower level of memory. It is also clear that the faster FLOP rates that the processors in HECToR and HPC-FF are capable of are

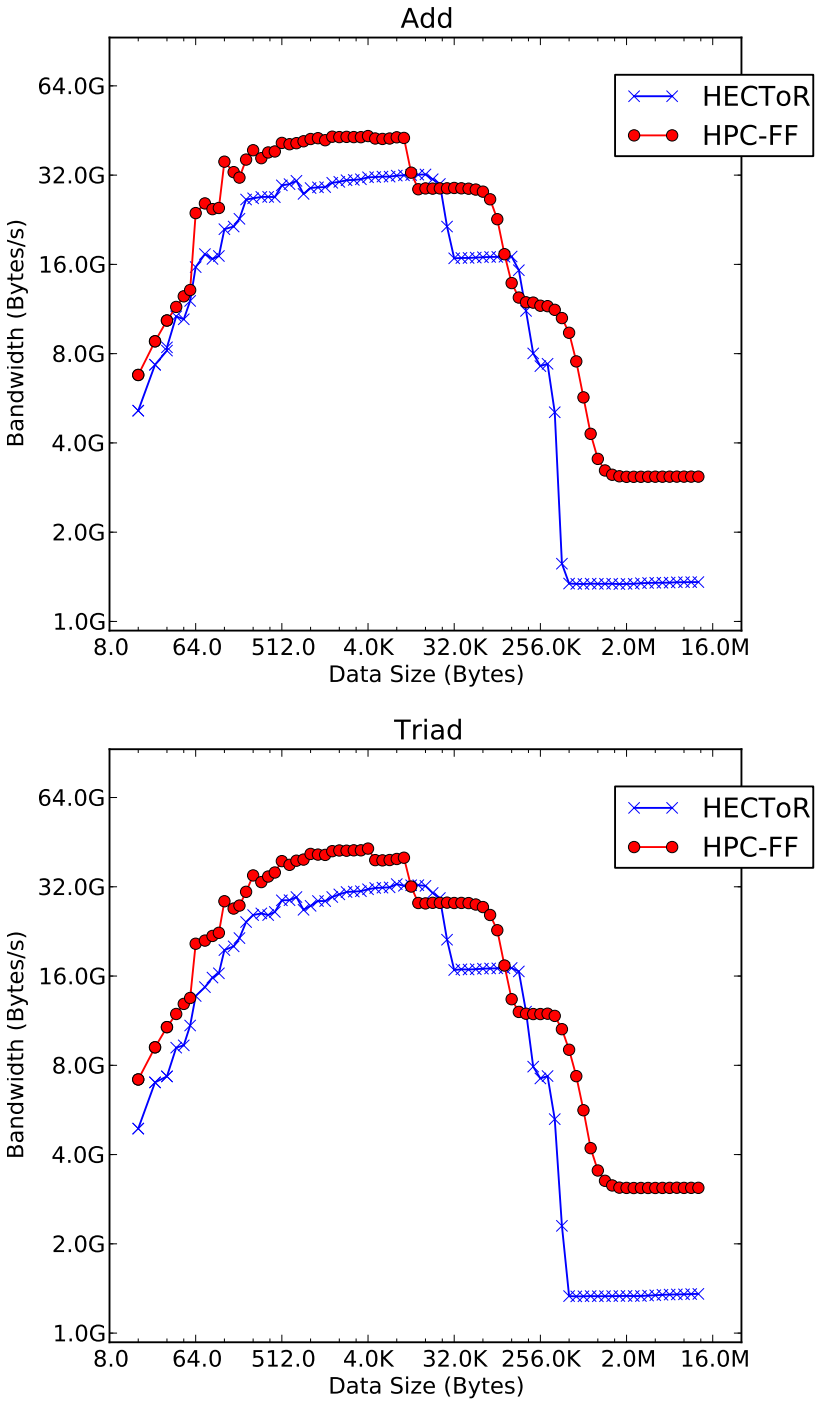


Figure 3.1: Results of the stream benchmark on a single processor for HECToR and HPC-FF.

only realised on arrays small enough to fit in the L1 cache, but large enough to overcome the overhead of the loop.

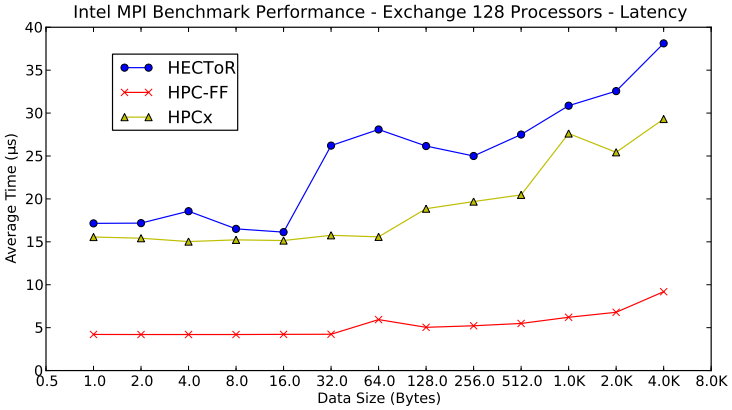
Ultimately the faster processor is the Intel Xeon X5570 processor used in HPC-FF, sustaining the highest bandwidth (performance) on both tests on small and large arrays. This is not unexpected as it is the most modern architecture and has the highest clock speed and greater bandwidth to memory of the two. Meanwhile the AMD Opteron processor in HECToR demonstrates a similar pattern of performance which it maintains for across its larger L1 cache, though ultimately its performance drops sooner than the Intel which has larger caches overall and greater bandwidth to main memory.

3.1.2 Network Performance

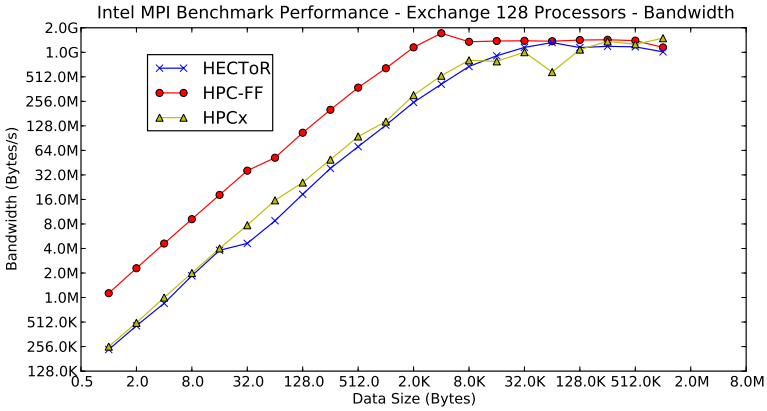
The second component of overall system performance is the raw performance of the network. Using the standard Intel MPI Benchmark[71] suite empirical values for the latency and bandwidth can be derived for similar operations performed in CENTORI. The suite of synthetic benchmarks is widely used to measure the raw performance of an interconnect and tests many different patterns of communication, however the most relevant benchmarks to CENTORI is the "Exchange" benchmark which performs a halo swap between processors organised in one dimension and the "All reduce" benchmark which performs a reduction across all the processors.

The tests were run on different numbers of processors, but results were taken from 128 processors which ensures that multiple shared memory nodes are being used and the internode performance is dominant.

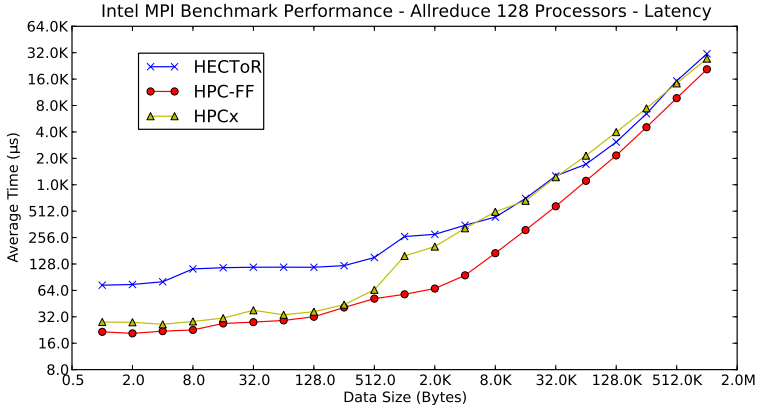
Figure 3.2a shows the average time to complete the "Exchange" of two messages between processors with data packets of increasing size. The data shows that with very small messages, below 4KB, the size of data the messages has little to no effect on the time to complete the transaction. This is the latency of the exchange dominating the communication time, from Figure 3.2a it is possible to estimate latency on each system. On HPC-FF it is $4\mu s$, on HPCx $16\mu s$ and on HECToR $17\mu s$ for this pattern of communication. As the message sizes increases, the latency becomes less dominant; the rate at which data is delivered tends to the aggregate bandwidth of the link between processors. Figure 3.2b shows the effective bandwidth of the link (a simple function of the packet size and the time taken to complete the transaction). On HPC-FF the bandwidth saturates at 1.5GB/s, on HPCx it continues to grow past 1.5GB/s and on HECToR saturates at 1.2GB/s. So, while each architecture has a similar level of bandwidth,



(a) Average Completion time vs data size for three architectures using 128 processors to run the "Exchange" benchmark.



(b) Bandwidth vs data size graph for three architectures using 128 processors to run the "Exchange" benchmark.



(c) Average Completion time vs data size for three architectures using 128 processors to run the "All reduce" benchmark.

Figure 3.2: Plots of results for HPCx, HECToR and HPC FF on the "Exchange" and "All reduce" benchmarks on 128 processors on each system.

the latency is significantly improved on the newest architecture HPC-FF, being four times better than the others.

The "All reduce" benchmark tests a different aspect of the MPI implementation's performance. This performs a reduction across all 128 processors and places the result on every single processor. This requires not only a fast hardware network and an efficient algorithm but also depends upon the topology of the network hardware. A simple implementation might evaluate a binary tree which for 128 processors would have 7 levels. As each level would require at least one message to other processors the minimum evaluation time would be 7 times the latency of an individual message. The latency on HECToR is $71\mu s$, HPCx $28\mu s$ and HPC-FF $21\mu s$, which are significantly below 7 times the internode latency for "Exchange" on HECToR and HPCx. There are multiple explanations for this behaviour: first, the value for the latency measured by "Exchange" has messages travelling in two directions which may affect the hardware's performance, whereas a tree evaluation passes messages in one direction only. Secondly the estimate does not account for the heterogeneous nature of the latency, messages passed intranode are likely to have much lower latency than internode messages. Therefore the larger the shared memory nodes, the fewer levels of the binary tree which operate internode. This could explain why HPCx which has a similar performance to HPC-FF in this benchmark despite having a much larger latency in the previous test. Each of HPCx's nodes are 16 way shared, compared with the 4 way share of HECToR.

These results are not an exhaustive study of the network performance of the systems, there are many options to tune the performance of the networks, many of which induce non-linear behaviour (eager vs rendezvous protocols). These tuning parameters would have an impact on the results as would running on a different number of processors. However, these values allow a more informed evaluation of CENTORI's performance when profiled.

3.2 Profiling CENTORI

Figure 3.3 is a profile of CENTORI running a simulation with a resolution of $128 \times 128 \times 128$. It shows the average (over 1000 time steps) total time spent by all processors on the various tasks during a single time step. CENTORI may run in parallel and can be split between a different numbers of processors which should reduce the wall clock run time of the simulation. A useful metric to meaningfully compare a code's performance on different numbers of processors is "cost". Cost is the sum total amount of processor time spent by all processors used on a

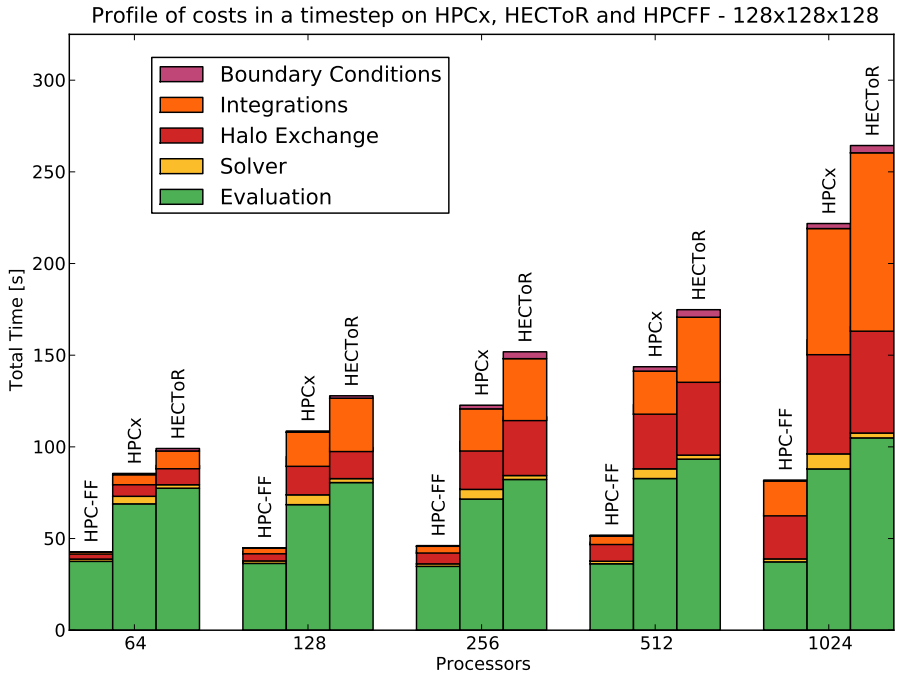


Figure 3.3: Averaged time spent by all processors performing 5 tasks during one time step of a $128 \times 128 \times 128$ simulation on HECToR, HPCx and HPC-FF.

particular task and is calculated from the measured wall clock time, T, and the total number of processors, N, as the total cost is the product, $T \times N$. A perfectly parallel application should see the cost remain constant across different numbers of processors because $T \propto \frac{1}{N}$. In reality CENTORI is not a perfectly parallel application and the overall cost will rise as more processors are used.

The total height of each column in Figure 3.3 represents the total cost of a time step on each processor architecture and for each processors count. For all architectures the total cost of a time step grows as the total number of processors increases, indicating that the parallelisation is not perfect and that some parts of the model cost more as the number of processors increase.

A note on CENTORI versions

During the course of this project CENTORI has undergone significant development work by the authors. To ensure a fair comparison for all optimisation work undertaken, all the results presented in this thesis refer to versions derived from a snapshot of the stable development tree

from the 22nd May 2009. All work compares like-for-like versions of CENTORI which perform the same fundamental equations. The results of the successful development work has been fed back into the main CENTORI trunk and is currently in day-to-day use.

3.2.1 Decomposing the performance

Looking at the individual components of the profile, Figure 3.3 shows the cost of 5 general tasks performed in CENTORI as measured by lightweight internal timers in the code. Three of these tasks, Boundary Conditions, Halo Exchange and Integrations, are dominated by network communication and the cost associated with each of these tasks can be seen to grow on all architectures as the number of processors increases. This result is expected as the time spent communicating is a function of the number of processors to communicate between. These tasks are the parallel parts of CENTORI.

The other parts of the profile are the Solver and the Evaluation which feature very little communication. If Equation 3.3 is a generalised differential equation solved by CENTORI (as described in Section 1.4.1), then Equation 3.4 is the generalised discretised solution in CENTORI with coupling in the radial dimension.

$$\frac{\partial \mathbf{X}}{\partial t} = \mathbf{f}(\mathbf{X}) \tag{3.3}$$

$$\mathbf{A}_i \mathbf{X}_{i+1}^{n+1} + \mathbf{B}_i \mathbf{X}_i^{n+1} + \mathbf{C}_i \mathbf{X}_{i-1}^{n+1} = \mathbf{f}(\mathbf{X}_i^n)\Delta t + \mathbf{X}_i^n \tag{3.4}$$

Equation 3.4 forms a tridiagonal matrix equation for each of CENTORI's core equations. This leads to two stages in evolving the core prognostic variables of CENTORI:

1. Evaluating the right hand side of Equation 3.4 which implies evaluating $\mathbf{f}(\mathbf{X}_i^n)$.
2. Solving the resulting tridiagonal matrix problem on the left hand side.

The profile shows that very little time is spent in the Solver on every platform, the majority of the time is spent in Evaluating the right hand side. The parallel communication components also form part of the evaluation algorithm, as all the parallel calls are from this component. It also shows that the cost of the serial computation grows only very slowly as the number of processors increases. This indicates that the serial components have been successfully parallelised and work is divided evenly, the only additional work for the serial component are operations performed on halo regions which increase in volume as the number of processors increase.

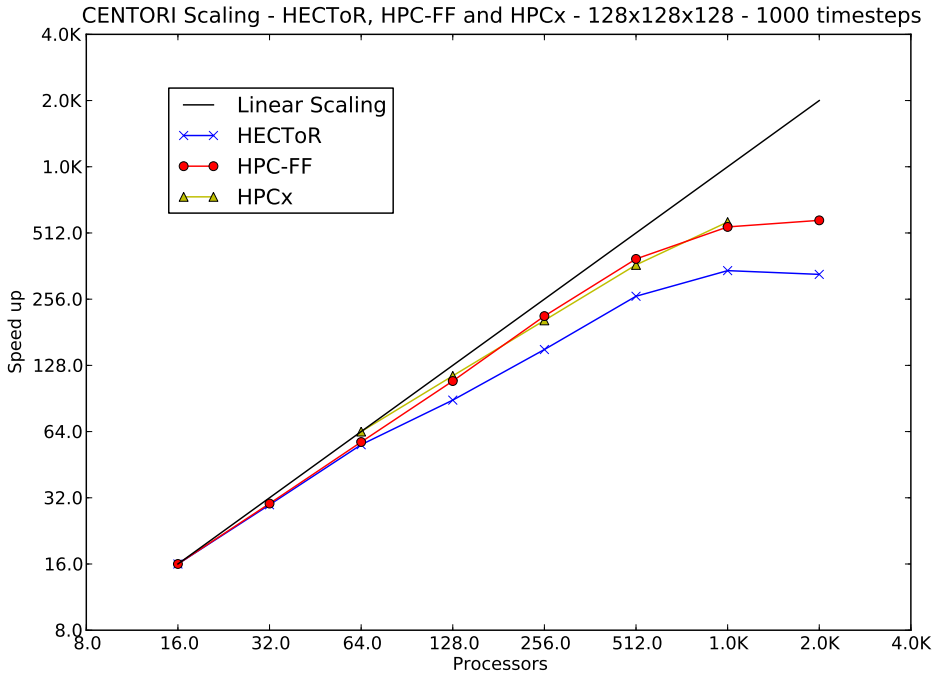


Figure 3.4: Scaling performance of CENTORI (from reference 16 processors) on HPCx and HECToR (vs linear scaling).

3.2.2 Scaling

Figure 3.4 is a scaling graph for CENTORI on each of the platforms between 16 and 2048 processors. It show the number of times faster the average time step of a $128 \times 128 \times 128$ resolution CENTORI simulation is for each number of processors versus the projected time for running the code as a serial task. A projection is used from 16 processors because CENTORI cannot be practically run at this resolution on any fewer processors. The graph also includes a line showing linear (perfect) scaling.

CENTORI continues to speed up on HPCx and HPC-FF for all processor counts tested, though the speedup continues to deviate further from linear scaling as the processor count grows. On HECToR the speed up decreases between 1024 and 2048 processors indicating that the model actually slows down. This shows that CENTORI has reached the limit of its scaling on this architecture at 1024 processors. It also shows that HECToR's speedup is less than that for HPCx and HPC-FF overall.

3.2.3 Comparing Architectures

Table 3.3 shows the raw data used to create Figure 3.3. It shows the processor time cost in seconds for each of the five profiled components on each processor count. Overall they show that HPC-FF is over 100% faster than both HECToR and HPCx on all processor counts and time steps cost between 1.9 to 2.7 times more on 1024 processors than they do on 64, showing the reduced efficiency of the code as the processor count increases.

On HPC-FF the cost of the serial sections remains almost constant as the number of processors increases (as expected for a parallel code), while it grows on HPCx and HECToR between 64 and 1024 processors but only by 27% and 35% respectively. The figures also confirm that HPC-FF's performance lead over HPCx and HECToR is reflected in the performance increases in the serial code as well. This is probably due to the increased floating-point performance and memory bandwidth of the Xeon processors as demonstrated in Section 3.1.1.

As the number of processors increases the proportion of the cost attributed to parallel code increases greatly. The parallel sections grow from approximately 20% of the total cost on 64 processors to approximately 60% on 1024 processors on HPCx and HECToR. HPC-FF again performs better, spending only 12% of its time on communication at 64 processors, rising to 53% on 1024 processors, which can probably be attributed to its better network performance as documented in Section 3.1.2.

3.3 Understanding CENTORI's Performance

CENTORI's performance can be separated into two major areas:

- Parallel communication
- Serial computation

Each component makes use of a different part of the computer's hardware. At the lower processor counts serial computation dominates which requires the code to be optimised to make use of the microprocessor architecture available. Chapters 4 to 7 cover in great detail the optimisation of CENTORI for the current and future generations of microprocessors. For CENTORI to scale to large numbers of processors the parallel communication needs tuning to optimise traffic over the interconnection network as parallel communication becomes a larger and larger part of the overall cost. Chapter 8 covers the analysis and tuning of CENTORI's parallel performance on HPCx, HECToR and HPC-FF.

# Processors	64	128	256	512	1024
Boundary Conditions	0.7	0.7	2.0	2.4	2.7
Halo Exchange	10.4	20.9	26.2	35.1	62.3
Integration	5.5	18.6	23.0	23.5	68.8
Solver	4.1	5.4	5.3	5.3	8.2
Evaluation	68.9	68.5	71.5	82.7	88.0
Total	89.6	114.0	128.0	149.0	230.0

(a) HPCx

# Processors	64	128	256	512	1024
Boundary Conditions	1.4	1.3	3.7	4.1	4.0
Halo Exchange	10.6	16.9	32.1	41.9	58.2
Integration	9.7	29.1	33.8	35.5	97.3
Solver	1.9	2.1	2.2	2.2	2.6
Evaluation	77.5	80.5	82.2	93.3	104.9
Total	101.0	130.0	154.0	177.0	267.0

(b) HECToR

# Processors	64	128	256	512	1024
Boundary Conditions	0.2	0.1	0.4	0.5	0.5
Halo Exchange	3.9	5.2	7.3	10.5	25.1
Integration	1.1	3.2	3.7	4.6	19.0
Solver	1.2	1.3	1.4	1.4	1.6
Evaluation	37.6	36.4	34.8	36.2	37.3
Total	44.0	46.3	47.5	53.2	83.5

(c) HPC-FF

Table 3.3: Total processor Costs in seconds of profiled components of a time step in a $128 \times 128 \times 128$ resolution CENTORI simulation.

Example 3.1 The original Fortran source for a vector "add" kernel.

```
integer, intent(in) :: n
real(kind(1.0D0)), dimension(n), intent(in) :: b, c
real(kind(1.0D0)), dimension(n), intent(out) :: a

do i=1,n
  a(i)=b(i)+c(i)
end do
```

Example 3.2 The original Fortran source for a vector "triad" kernel.

```
integer, intent(in) :: n
real(kind(1.0D0)), intent(in) :: s
real(kind(1.0D0)), dimension(n), intent(in) :: b, c
real(kind(1.0D0)), dimension(n), intent(out) :: a

do i=1,n
  a(i)=b(i)+s*c(i)
end do
```

3.4 Practical Consequences for Performance

While understanding the design of a microprocessor is very useful, it is difficult to understand what effect each factor will have on a real world code as the overall performance results from the complex interaction between components. HECToR and HPC-FF are both based on the x86-64 instruction set, the dominant architecture in the Top 500 and on workstations. As the processors can use the same instruction set, it is possible to compare exactly the same sequence of instructions on both processors. By generating assembly code for an "add" kernel (Example 3.1) and a "triad" kernel (Example 3.2) with each of the compilers (see Table 3.4) , the result can be built into the executable on each system. The code executed on each processor is the same (there is direct mapping between assembly code and machine code) and so this allows a true like-for-like comparison between systems and between compilers.

Vendor	Version	Options	Native
Portland (PGI)	8.0-5	-fastsse -O3	Opteron
Pathscale	3.0	-O3 -OPT:Ofast	Opteron
Cray	7.0.4	-O3	Opteron
Intel	10.1	-O3 -msse3	Xeon
GCC	4.4.0	-O3	Opteron
PGI (NoSSE)	8.0-5	-fastsse -O3 -MVect=nosse	Opteron

Table 3.4: The compilers tested, with versions and arguments used.

The overall test harness passes a single test subroutine three arrays of constant size and is capable of passing the subroutine arrays that are either aligned on 16 byte boundaries or unaligned which allows the effect of data alignment to be taken into account. The subroutines are run multiple times to allow data to move to the appropriate cache level and then the subroutines are run many times so the timing is averaged to find the time taken to execute the result. This is converted into an overall bandwidth by calculating the amount of data transferred into and out of the processor and the time taken to do this.

3.4.1 Comparing processor performance

The first test is a direct comparison between the processors. By comparing the results for the Cray compiler only it is possible to see some of the features described above affecting the performance of the processors. Figure 3.5 shows the results of these tests for a variety of input arrays with different lengths. Some features to note are:

- It is easy to see the steps where the size of the input arrays passes outside the size of the caches. These kernels offer no opportunity for cache reuse, so unless the data is already on the cache (from a previous iteration of the test) the data has to be fetched from the lower memory level either as the result of a pre-fetch request or upon issuing the instruction. On both processors it is possible to identify 4 plateaus that corresponds to the separate levels of the memory hierarchy, the Level 1 cache, the Level 2 cache, the Level 3 cache and the main memory system.

- Though the smallest arrays ($<$ 128 bytes) must clearly operate on the highest levels of cache, their performance is relatively poor. This is due to the overhead inherent in the subroutine (the loop counter etc.). Proportionally this is much larger when operating on small arrays and the peak performance is seen on array sizes which occupy much larger portions of the highest level cache.

In both processors there are separate pipelines for multiplication and addition. This potentially limits the maximum bandwidth achievable by the Add kernel compared to the Triad kernel. On the Opteron it can process two adds every cycle (at 2.3GHz)[55], which is equivalent to:

$$2\frac{\text{FLOP}}{\text{cycles}} \times 2\frac{\text{Word}}{\text{FLOP}} \times 2.3 \times 10^9 \frac{\text{cycles}}{\text{s}} \times 8\frac{\text{bytes}}{\text{word}} \approx 69\text{GBs}^{-1}$$

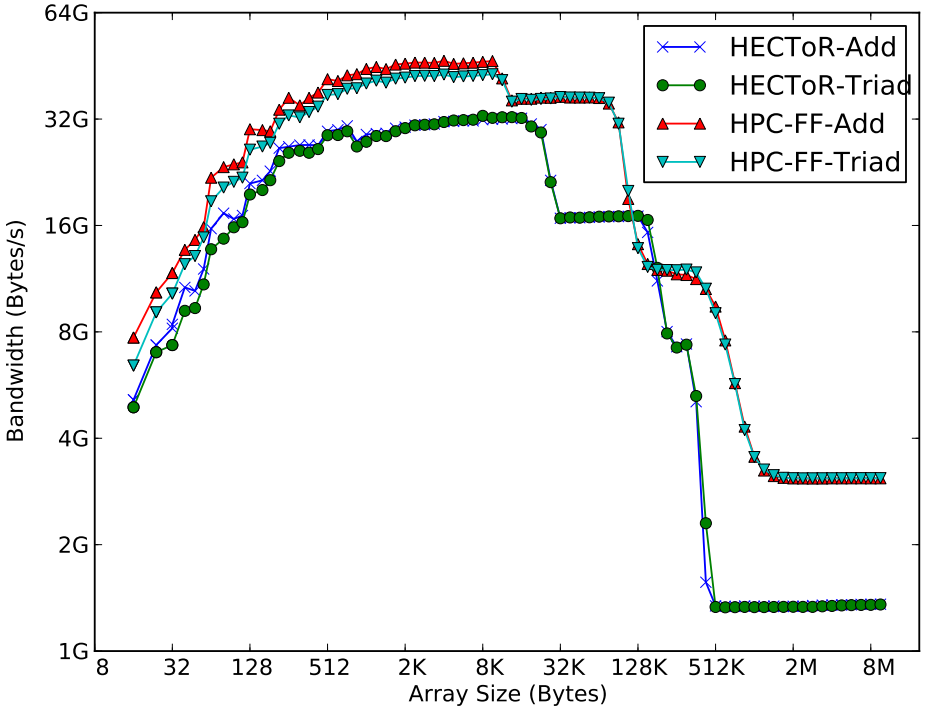


Figure 3.5: Performance of each kernel when compiled using the Cray Compiler and run on the AMD Opterons in HECToR and the Intel Xeons in HPC-FF.

where $1\text{GB} = 2^{30}\text{B}$, whereas the peak processing rate was measured at 33GBs^{-1} which is 48% of the theoretical peak performance. The Xeon can similarly process two adds every cycle[72], though at a higher clock speed of 2.93GHz which gives it a theoretical peak performance of

$$2\frac{\text{FLOP}}{\text{cycles}} \times 2\frac{\text{Word}}{\text{FLOP}} \times 2.93 \times 10^9 \frac{\text{cycles}}{\text{s}} \times 8\frac{\text{bytes}}{\text{word}} \approx 87\text{GBs}^{-1}$$

Its peak recorded rate was 42GBs^{-1} which is also 48% of the peak performance.

Now looking at the Triad kernel, this allows the introduction of the multiplication pipeline in each processor which doubles the rate (theoretically) at which data can be processed. Both HECToR and HPC-FF achieve almost exactly the same aggregate performance as the Add kernels. This suggests that the limit on performance (bottleneck) in these streaming kernels is not the performance of the floating-point unit, but the ability of the L1 cache to transfer data to and from it. Reversing the calculation we see that the peak bandwidth on HECToR:

$$\frac{33\text{GBs}^{-1}}{2.3\text{GHz}} = 15.4\frac{\text{B}}{\text{cycle}}$$

and HPC-FF

$$\frac{42\text{GBs}^{-1}}{2.93\text{GHz}} = 15.4\frac{\text{B}}{\text{cycle}}$$

This rate is just below performing two double precision floating-point values loads or stores per cycle, which is insufficient to keep either floating-point unit saturated. Therefore many more floating-point operations can be performed on data in registers than can be performed from data in the L1 cache.

Main Memory Bandwidth

One of the distinct differences between the architectures is the bandwidth to main memory. Once the kernels approach 1MB in size and over, the caches of each of the processors are too small and all the data must be streamed from memory into the caches. The resulting rate is a practical measure of the main memory bandwidth, which is a product of the overall processor bandwidth for the DRAM technology (e.g. DDR2-800 on HECToR and DDR3-1066 on HPC-FF) shared between each of the processors connected to the memory bus (four processors in each case as each is part of a quad core package). The measured memory bandwidth on HPC-FF is 3.3GBs^{-1} per processor and on HECToR 1.5GBs^{-1} per processor; this increase in memory bandwidth and the improved clock speed explains the increased performance of CENTORI on HPC-FF in computation sections as recorded in Section 3.2. In these sections the computation took half the time on HPC-FF compared to HECToR with only a 27% increase in clock speed which indicates that *CENTORI's computation is dominated by the memory bandwidth of the architecture it is run on.*

3.4.2 SIMD instructions

Both of the processors are able to issue Streaming SIMD Extensions (SSE) instructions, which include small vector (SIMD) instructions allowing two double precision floating-point operations to be issued as a single instruction. These instructions can be completed once every cycle, doubling the floating-point rate of the processor; however they are restricted to loading and storing data in pairs aligned along 16 byte boundaries in memory (i.e. assuming byte level addressing the address is divisible by 16). If both operands are similarly aligned the data can be processed in the most efficient way, however if the data is incorrectly aligned (i.e. the data is on the 8th or 24th byte boundary) there may be a performance penalty as the operand cannot

be loaded with a single instruction and instead two loads must be issued. SSE instructions also update standard x87 floating-point instruction set with more modern equivalents which require fewer instructions compared to their predecessors.

General Use

To attempt to quantify the real world performance benefits that SSE instructions provide, a version of the code that does not include any SSE instructions was produced using the PGI compiler. This was compared against the version of the code with the best performance, in this case the version with the Cray compiler which does include SSE (including SSE SIMD) instructions making it possible to measure the difference in performance. Each test operates on data structures which are aligned on 16 byte boundaries by passing input and output arrays that have been modified to be aligned, thus theoretically allowing full use of the SSE SIMD instructions. Figure 3.6 shows the performance comparison of each of the kernels.

The inclusion of SSE SIMD instructions theoretically doubles the potential floating-point capacity of the processors and from the results it can be noted that the performance is most severely affected when the operands are held in the highest cache levels on both processors. However, inclusion of SIMD instructions seems to equate to a peak increase in performance of 1.5 times on HECToR and just under 1.4 times on HPC-FF as seen in Figure 3.7. This is likely a result of the Load Store Unit in each processor only being able to issue limited numbers of read and write operations and only in certain combinations[55]. This restriction on the flow of data to the floating-point units is an explanation for the limited speed up. A second feature to note is that the performance is only enhanced in the Level 1 and Level 2 caches where the memory bandwidth restrictions are less severe, there is virtually no speed up (and apparently some slow down) when operating on the Level 3 and Main memory on both processors.

Alignment

The restriction on the use of SIMD instructions that data be aligned on 16 byte boundaries is more difficult to control in CENTORI. Fortran, unlike C, does not provide, in the standard, a method of obtaining aligned data during an allocate or from automatic arrays. This means the programmer is at the mercy of the kernel (which in the main does provide aligned memory), however what is the effect of working on arrays which are unaligned - how much performance can be expected to be lost? Figure 3.8 shows performance figures when the kernels created by

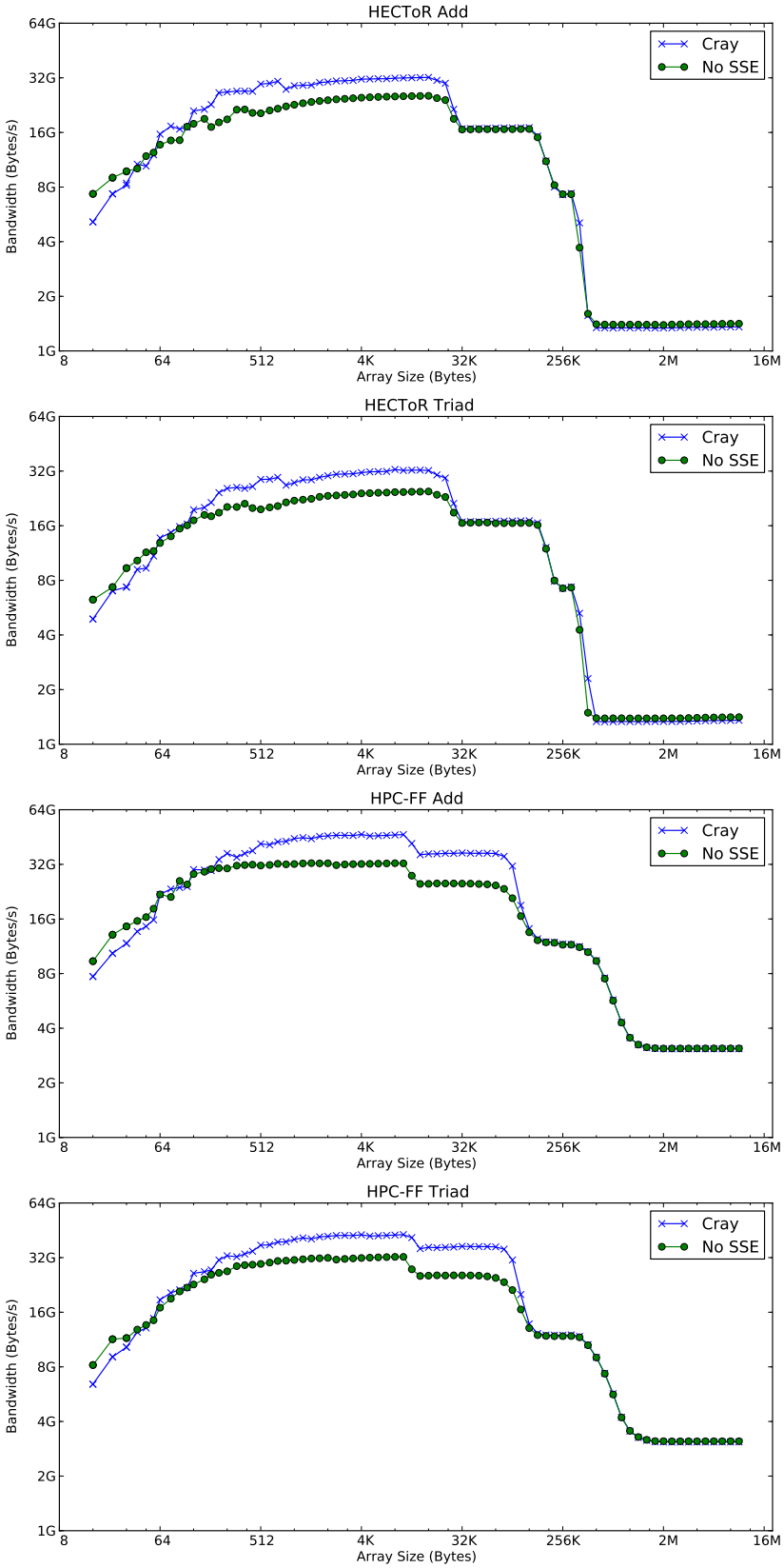


Figure 3.6: Comparing the performance of the kernels compiled by the Cray compiler and kernels which do not issue SSE instructions.

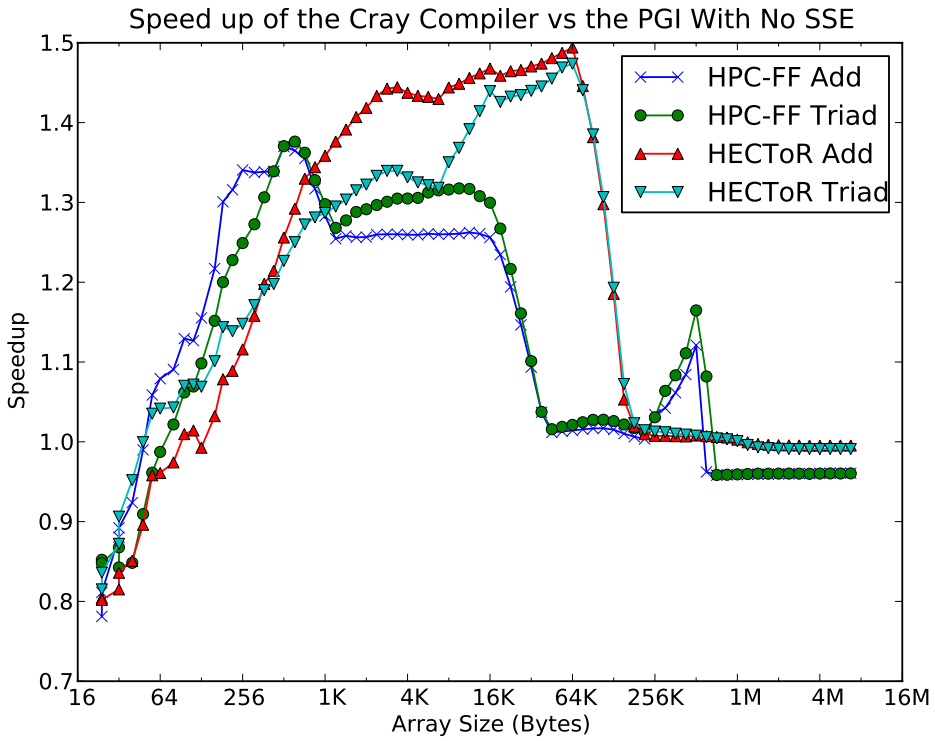


Figure 3.7: Relative speed up of the kernels over No SSE instruction PGI compiler and the Cray compiler.

the Cray compiler are given differing alignments of data, the alignments are labelled for arrays `a`, then `b`, then `c`, with `0` representing aligned data and `1` unaligned data, e.g. `010` expresses `a` and `c` as being aligned and `b` unaligned.

It is clear from this figure, that as in the previous section, SIMD instructions have no effect when operating on main memory or on the lowest levels of cache, the bottleneck is not the alignment but the overall bandwidth of the system. Figure 3.9 shows the speed up of the kernels operating on completely aligned data versus a kernel operating on data which is misaligned (where the `a(1)` and `b(1)` are not aligned on a 16 byte boundary whereas `c(1)` is aligned on a 16 byte boundary).

There are significant performance differences due to alignment of up to 1.4 to 1.6 times on three of the processor and compiler combinations, with a maximum value of 2.3 between this ideal and worst case scenario for PGI on the Triad on HECToR.

In nearly all cases the compiler has no guarantees about the alignment of the arrays that are being passed to the subroutines in question. As the subroutine must operate in all circumstances, the compiler must generate code that will operate in a worst case misaligned scenario. It may however generate different versions of the subroutine which are picked depending upon the alignment of the input operands, allowing for an optimal case and for other suboptimal cases. The number of versions that a compiler produces is implementation dependent and part of the internals of the compiler, however it is possible to identify the clustering of the performance profiles to identify where the compiler is executing similar versions (the performance is similar). With the Cray compiler it is possible to identify, from the similarity in performance, four possible groups of performance which indicate the compiler has generated 4 different cases, however groups are not necessarily the same on each processor, indicating there are more complicated reasons for the differences in performance than versioning alone.

Similarly, it is key to note that the performance impacts of alignment can only be seen with arrays that are small enough to fit into the higher levels of cache (L1 and L2 in this case).

3.4.3 Compiler Performance

Five compilers were used to generate code for the kernels specified, plus the additional PGI compiled version with no SSE instructions. This allows some comparison in the choices made by the compiler in these limited cases. Figure 3.10 is a direct comparison of the compiler performance for each kernel on each platform when operating on completely aligned data for

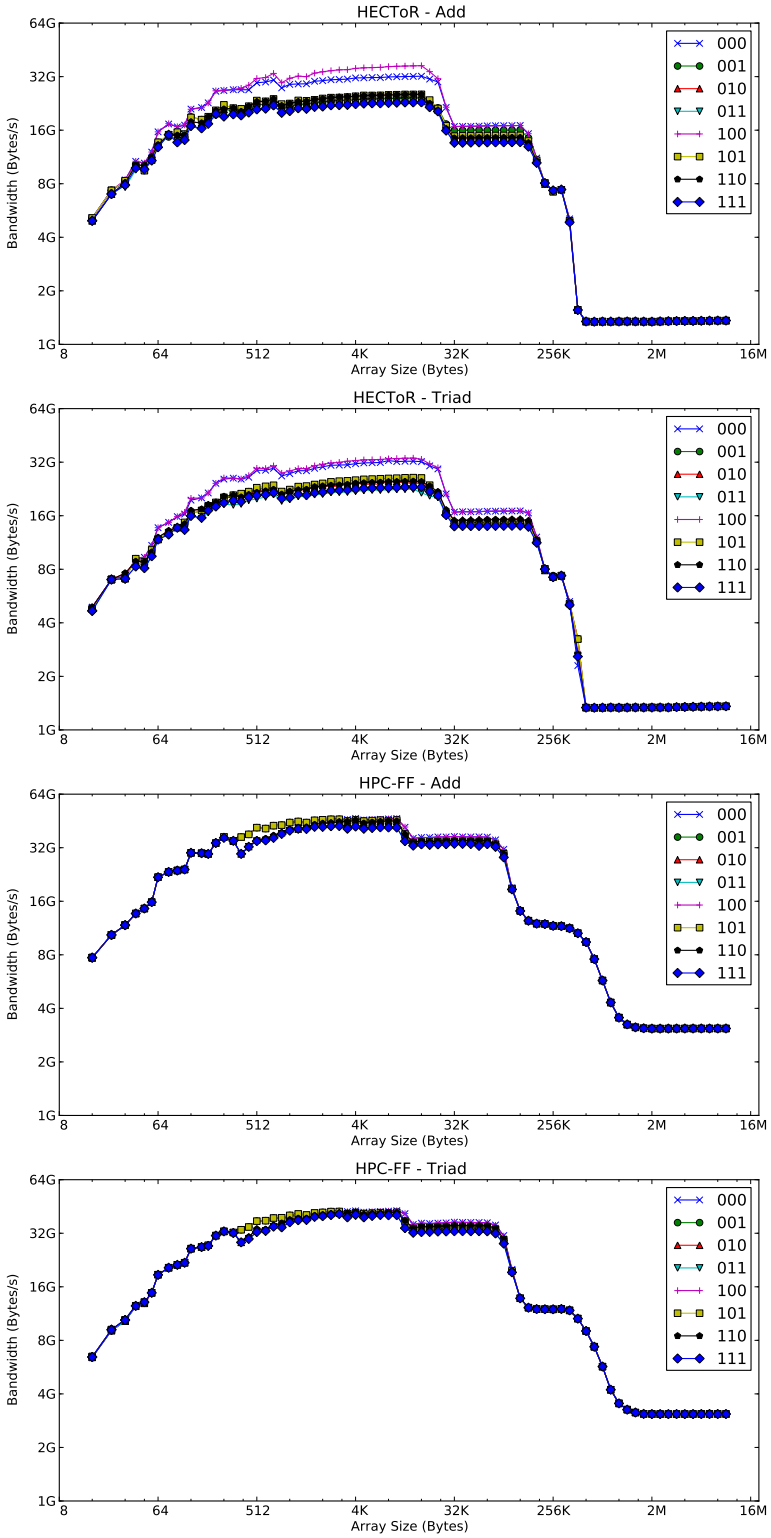


Figure 3.8: Performance of the kernels compiled by the Cray compiler when operating on data with arrays which have different 16 byte alignments. (Legend codes arrays ABC, 0 aligned and 1 unaligned).

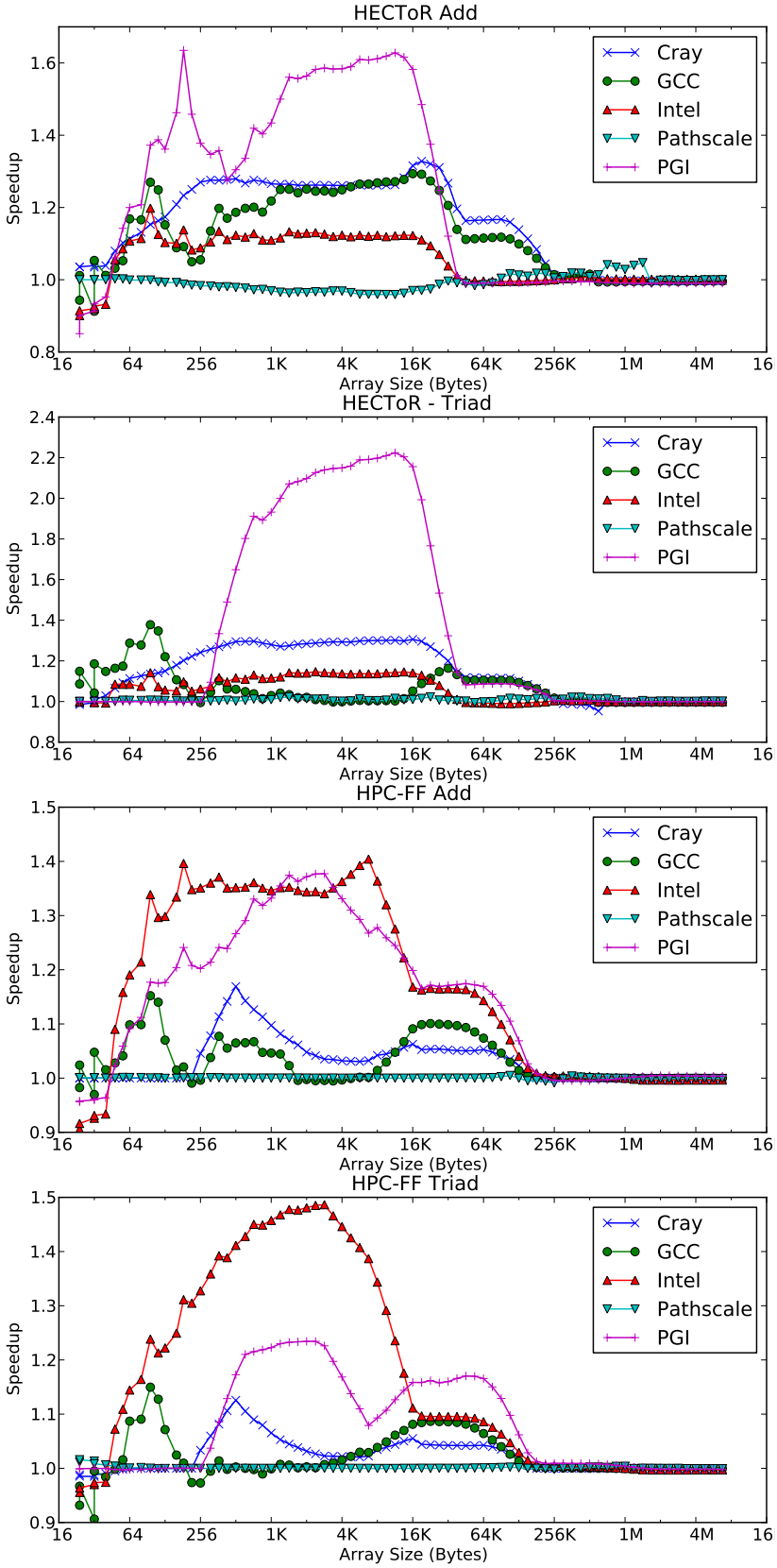


Figure 3.9: Speed up of aligned (000) input over misaligned input (110) averaged over multiple values for the kernels on each architecture for each compiler.

all three input arrays.

In a common theme for all of the parameters that have been studied, the effect the compiler has on performance is limited to the highest memory levels (i.e. Level 1 and Level 2 cache). In these regions, all the compilers produce good improvements over the NoSSE version with the Cray and Intel compilers performing the best over all the tests and the PGI compiler performing very well on HECToR with the Add kernel. Ignoring the No SSE compiler, the PathScale compiler produces the most consistently poor results, being beaten by the open source GCC compiler (which is usually not considered a high performance compiler) in three of the four tests.

PGI Compiler Performance in Main Memory

Figure 3.10 shows that all the compilers converge on the same performance when operating on arrays which are over 2MB in length except for the PGI compiler. It is clear from these results that the PGI compiler is able to achieve performance which is approximately 20% faster on HPC-FF and 50% faster on HECToR than all the other compilers. Testing shows that the results are numerically correct, and testing using hardware counters on HECToR (see Section 3.6.2) show that the code suffers from fewer Idle Floating-Point Unit exceptions. This implies there were fewer stalls when waiting for an instruction fetch than on code from other compilers. However it is unclear if this is a cause or a symptom of the performance advantage.

3.5 Conclusions about Serial Performance

These synthetic benchmarks are not universal measures of performance for the processors, nor can they be extrapolated to provide all the information about how a real code will perform on the processors in real life. There are many complicated factors that cannot be covered in this test, since codes like CENTORI contain memory usage patterns and sequences of calculation which are far more sophisticated. However these simulations do show some key issues to consider when optimising a code like CENTORI:

1. Avoid loading from main memory wherever possible, data that is held in cache is accessed far more quickly and this has a significant impact on performance. Whatever can be done to improve the cache reuse of data, or to improve the streaming of data into cache, the better the effect on performance.

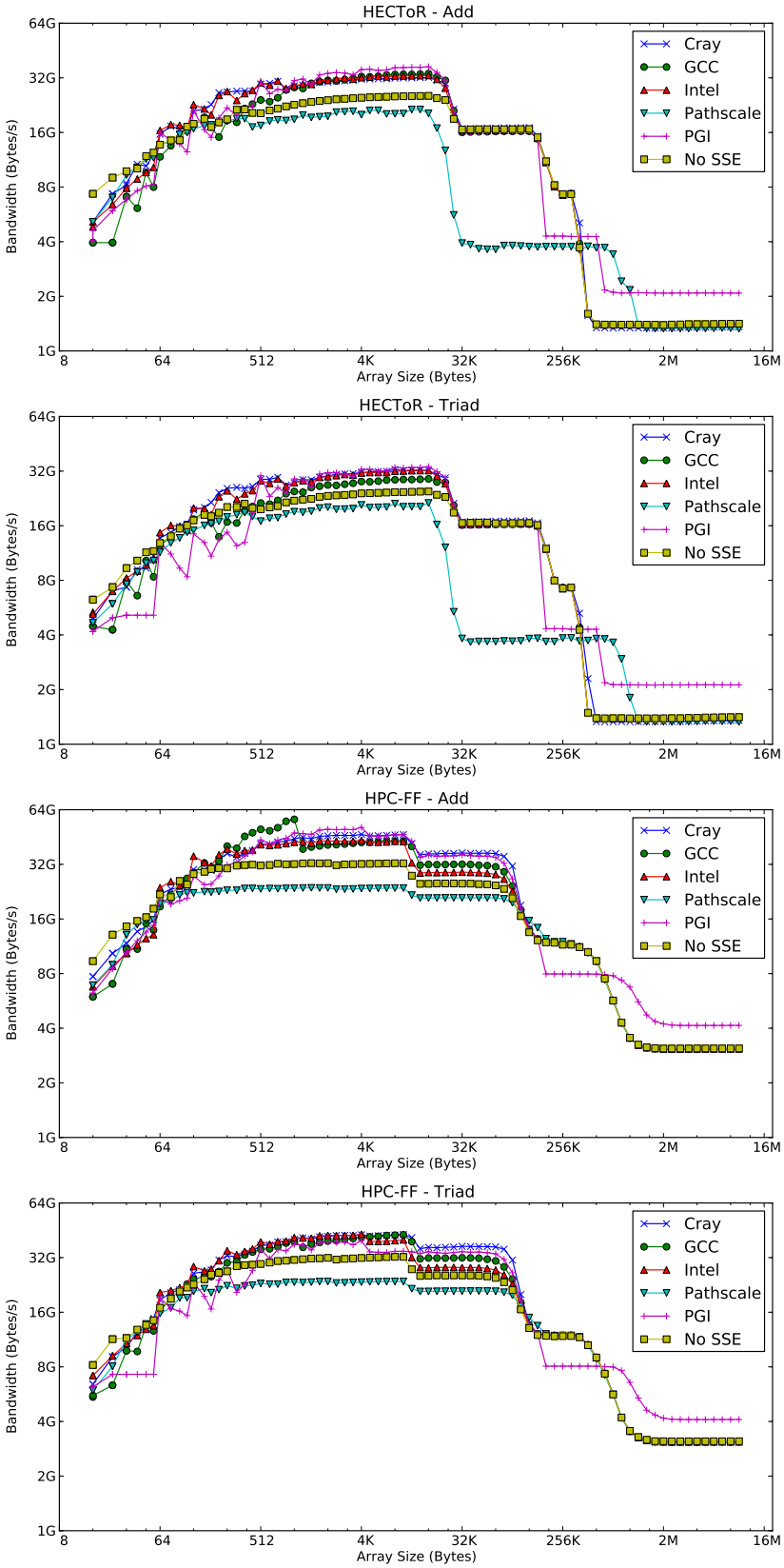


Figure 3.10: Direct comparison of the performance of compilers from the same source code for "triad" and "add" kernels on HECToR and HPC-FF.

2. Code should be written to make simultaneous use of differing floating-point units where they exist (i.e. Fuse Multiply Add in Power Architectures or separate Addition and Multiply units in x86 processors). This is not always possible, however scale operations could be fused with additions etc.

3. Improvements in performance from compilers, SSE instructions and alignment are only relevant when the data is available on the highest cache levels L1 and L2. Optimisation should first concentrate on moving data from main memory into cache wherever possible, then applying optimisations for SIMD.

4. Vectors have to be of sufficient length to counteract the overhead of loops and to allow the pipelines to become saturated (i.e. vectorise) however not so large that the first results are evicted from cache before being reused. This implies that vectors have to be longer than 512 to 1024 bytes in order for the full bandwidth to become apparent and less than 32 KBs to stay on the L1 cache. Branch statements should be removed from inner loops to avoid interrupting the vectorisation and to prevent stalls in the pipeline, wherever possible branch statements should be moved to the outermost loops.

5. Floating point operations like square root and division should be avoided wherever possible as they are completed at a much slower rate than multiplication, addition and subtraction operations. This is a very common optimisation provided by compilers, but should influence general programming style.

This section is a justification of these principles, which will feed into the design of a secondary set of benchmarks which test the performance of compilers and hand optimisations on more realistic operations used by CENTORI in Chapter 4. Though these experiments were only conducted on x86 processors used in HECToR and HPC-FF the advice is widely applicable to other scalar architectures like the Power 5+ used in HPCx or Vector architectures like the NEC SX-9 or Cray X2.

3.6 CENTORI's Serial Performance

In order to understand how CENTORI's performance has changed with the changes made during this project the original performance characteristics of the code should be presented. This section offers such an analysis of a baseline version of CENTORI which has not been

Operation	% Serial Processor Time
Misc Evolution	32.7%
Update Vectors	19.0%
Curl	8.0%
Add	6.3%
Dot	5.6%
Scale	4.7%
Cross	4.0%
Subtract	3.9%
Divergence	3.2%
Gradient	3.1%
L2 norm	2.9%
Solver	2.2%
Other	4.3%

Table 3.5: Profiling results for $128 \times 128 \times 128$ resolution CENTORI simulation on HECToR on 64 processors.

specifically optimised with the techniques documented later in this thesis.

3.6.1 Measuring CENTORI's Serial Performance

Table 3.5 presents a more detailed profile of where CENTORI spends time while evolving the prognostic fields than reported in Section 3.2 . The figures are the averages over 100 time steps gathered on HECToR for a $128 \times 128 \times 128$ simulation decomposed over 64 processors arranged in a $4 \times 4 \times 4$ grid. These components account for 79% of total run time and form serial operations; parallel operations are not included which account for the remaining 21% of total run time.

These profiling figures were obtained using the Cray Performance Analysis Toolkit (CrayPAT), a proprietary application for instrumenting and analysing executables on Cray XT5 architectures like HECToR. The values were acquired using sampling which has only a small impact on the overall run time (run times increased by 1% while being sampled).

The profile time is attributed to the various different tasks performed by CENTORI: operations like curl of a vector field, adding two vector fields, performing a cross product are represented. However, CENTORI's central algorithm is implemented in a large subroutine which calls many of the other subroutines listed but also performs a lot of calculation directly. Time spent on operations in this top level subroutine is labelled as "Miscellaneous Evolve".

From this profile 61% of the processor time can be attributed to a set of 10 well defined mathematical operations and of the 33% of time spent in "Misc Evolution" a significant proportion is attributable to operations performed over local scalar quantities that have been coded

Operation	L1 % Hit	L1 HPM	L1+L2 %Hit	L1+L2 HPM	% FP Peak	% Vector
Misc Evolution	96.9%	4.03	97.5%	5.09	1.3%	13%
Update Vectors	98.9%	10.92	99.4%	22.32	1.9%	0%
Curl	95.1%	2.56	96.3%	3.36	0.8%	55%
Add	99.6%	33.72	99.6%	33.90	0.6%	0%
Dot	97.0%	4.16	97.0%	4.23	0.6%	0%
Cross	99.3%	17.86	99.4%	20.71	2.2%	0%
Subtract	99.6%	34.99	99.6%	35.05	0.6%	0%
Divergence	96.0%	3.14	97.4%	4.72	1.0%	93%
Gradient	96.8%	3.96	98.2%	7.10	0.7%	84%

Table 3.6: Profiling results for $128 \times 128 \times 128$ resolution CENTORI simulation on HECToR on 64 processors.

explicitly at the top level. All of the 10 operations and those in "Misc Evolution" operate on the same data structures and are closely coupled, for example the input to an "Add" operation may be the output of a "Curl" or any other operation. Any attempts at optimisation would involve overhauling large sections of the code but any improvements would be to code that currently accounts for 94% of run time. The tridiagonal solvers, which might have been expected to be expensive, only contribute to 2.2% of runtime, so optimisation of these routines is not a priority.

3.6.2 CENTORI's Interaction with the Processor

So how well is CENTORI using the processor hardware? Is there room for CENTORI to be optimised? The CrayPAT tool also allows access to performance counters available in the microprocessor hardware. These record details about the microprocessor's operation that are usually unavailable, such as the total number of vector floating-point operations issued or the number of times the L1 data cache was accessed. These figures can be collected for individual subroutines using the "trace" method which alters the executable and adds extra instructions which keep track of the run time statistics. This additional instrumentation, when included in every subroutine, would cause too much interference to the overall run time so only a subset of the subroutines were traced (run time increased by 2% while being traced).

Table 3.6 shows the results of tracing CENTORI using the hardware performance counters. Only a relevant subset of all possible counters are recorded. The quantities presented (derived from raw hardware counters) are:

- *L1 % Hit* - The percentage of references to the Level 1 Data Cache where the data is present, i.e. a "hit". This figure should be as close to 100% as possible as the remaining references were misses.

- *L1 Avg HPM* - The average L1 hits-per-miss i.e. the average number of times a double precision floating-point location was accessed in the Level 1 cache before there was a miss (i.e. the data was referenced and not found in the cache so had to be retrieved from main memory). This figure is derived from the L1 hit ratio.

- *L1+L2 % Hit* - The percentage of references to the Level 1 Data Cache or the Level 2 Data cache that result in a hit.

- *L1+L2 Avg HPM* - The average L1+L2 hits-per-miss i.e. the average number of times a double precision floating-point location was accessed in the Level 1 or Level 2 cache before there was a miss. This figure is a reformulation of the L1+L2 hit ratio.

- *% FP Peak* - The percentage of the theoretical peak double precision floating-point performance achieved (peak is 9.2×10^9 FLOPs).

- *% Vector* - The percentage of double precision multiply and add operations which were completed as vector SSE instructions.

The results are measurements of the performance of the whole architecture, not just the performance of the instructions in the executable. For example if the hardware was to pre-fetch a value successfully into the L1 cache it will increase the % Hit rate, even if the reference is the first in the program (which would normally be a compulsory cache miss). Similarly, the hit-per-miss ratio is a measure of the number of times the L1 cache had the right data at the specific location, not the number of times an individual piece of data was accessed before being evicted.

The figures show a wide variation in the performance of individual pieces of CENTORI. Section 3.4 has demonstrated that the processor performs best when operating on data that is in one of the processor caches (preferably L1).

The L2 cache is a victim cache for the L1 (i.e. data has to pass through the L1 cache before it is evicted into the L2 cache), when there is little or no increase in the hit ratio between the L1 cache and the combined L1+L2 caches it confirms that the L2 cache is having almost no effect on the performance. This indicates that very little data is being reused once it leaves

the L1 cache which implies that the code is not reusing the data, but the prefetcher is doing a good job of predicting the usage patterns. The Add and Subtract operations have very high cache hit ratios (99.6%) and corresponding hits-per-miss values (33-34) which would normally imply good cache reuse. However there is no corresponding increase in the L1+L2 cache hit ratio and the codes are known not to reuse data, performing only one operation on each piece of data which suggest the high ratio is the result of successful pre-fetching by the software and/or hardware. The simplicity of the sequential memory access patterns makes prediction very easy and it appears the data is being successfully placed in cache before use. However, it is clear that the performance is still bound by memory bandwidth rather than the floating-point units with the peak performance only 0.6% of peak (actually 1.2% of peak Add performance).

The Curl, Divergence and Gradient operations have much poorer cache hit ratios (96.3%, 97.4% and 98.2%) than the Add and Subtract routines which inhibits their overall performance. This is partially a function of the internal vector data types being stored as a nine component vector while only three of the components are read or written during each operation, being spatially local the three components will be on the same cache line as some of the unneeded data which will be loaded into the cache (see Section 4.1.2). There are signs that data is being reused as the L1+L2 cache hit ratio is larger than the ratio on the L1 alone. However the overall percentage of peak floating performance for operations which feature a large number of floating-point instructions is still weak at 1.0% and less.

In between, operations like Update Vectors and Cross product show reasonable cache hit ratios (99.4%) and show the best percentages of floating-point performance of any operations, however at 1.9% and 2.2% are still relatively poor results on this architecture. The computational intensive kernels and use of every element of the nine component vector data type help improve the use of cache and make good use of the floating-point hardware.

The remaining evolution code has a poor cache hit ratio overall and mediocre floating-point performance. Analysis and speculation of the reasons for this are difficult as the code is a mixture of many different operations, however overall any work to improve cache reuse on this section of the code would prove beneficial.

Overall vectorisation seems to have been suppressed in the majority of operations, no vector instructions (packed SSE) were issued by most operations. However the computationally intensive operations, Divergence and Gradient, do appear to be including a significant proportion, with Curl performing just over half. Unfortunately, the poor cache usage ratios of these

operations means the advantage of the vector instructions is probably being lost to the time spent waiting for cache misses to be resolved.

3.6.3 Optimising CENTORI

The figures from the hardware performance counters show that CENTORI is not performing optimally on HECToR. Though this alone is reason enough to optimise CENTORI it is likely, due to the similarity of the architectures, that CENTORI has room for improvement on HPCx and HPC-FF as well. The results show that CENTORI could make better use of the caches and improve its overall floating-point performance as a percentage of peak. Inspection of the existing code and analysis of the figures suggest that part of the problem with performance is due to the layout of CENTORI's data structures in memory which should be investigated and altered in order to improve its performance. Chapter 4 documents this investigation and its results and recommendations.

Chapter 4

Optimising CENTORI's Memory Structures

Chapters 2 and 3 describe the major performance features of modern MPP architectures and the performance of a few simple kernels that crudely model the memory access patterns of modern scientific applications like CENTORI. Section 3.5 describes a few simple heuristic arguments for achieving performance on these modern architectures and profiles CENTORI's performance using these "rules of thumb" as a guide.

4.1 What can be Changed to Improve CENTORI's Serial Performance?

CENTORI's serial computation is dominated by the calculation of the right hand side of the plasma evolution equations rather than the solution of the resulting tridiagonal matrix. Vector calculus operations like curl, divergence and gradient as well as standard operations like addition, multiplication and cross products dominate the profiling analysis of CENTORI in Section 3.5. The nine vector algebra operations identified in Section 3.6.1 account for 42% of total run time, updating the components of vectors in each coordinate system takes a further 19%. Optimising these routines should yield significant improvements in overall application performance.

4.1.1 Reduce the number of subroutine calls

CENTORI is a modular application that has been divided into small subroutines making it easier to maintain and develop but inhibiting its serial performance. Each of the subroutines is quite simple and more complex actions are built from repeated calls to these subroutines. This approach is inspired by object oriented design which has successfully sustained the quality and maintainability of the code, but introduces many jump and branch statements that need to be evaluated before the flow of the program can continue. Interruptions like these may inhibit the compiler from issuing SIMD instructions and disrupt the program's execution potentially causing pipeline stalls that reduce the overall throughput. Whilst some of these problems can be addressed by the processor using features like speculative execution and branch prediction or compiler optimisations, like function in-lining and inter-procedural analysis, the potential for poor performance remains.

Despite the modularity of the code, there are very few instances where operations are performed on a per grid point basis. Most operations are functions of the larger field structures and are implemented by calling the same subroutines for each element of the field. Some improvement could come from removing or replacing subroutine calls with in-lined code which has been shown to produce performance improvements in many applications[73]. Comparison of applications written in C (a procedural language like Fortran) and C++ (a true object oriented language) show that object oriented programs generally have much deeper call stacks; an indication of large numbers of nested subroutine/function calls[74]. A more radical approach would be to implement optimised versions of the field operations with much longer loops and fewer branches and so improve the overall performance.

4.1.2 Reduce the size of vector data types

CENTORI stores nine double precision floating-point numbers and 1 integer for every vector data point. These represent the three components of the vector in each of the three different basis sets plus an integer to track which values are up to date. However, only one set of components in a single basis set needs to be stored as other representations can be computed from any other set. This would result in CENTORI requiring significantly less memory overall and reduce the amount of data processing by operations that are indifferent to the representation being used (like addition, subtraction, multiplication).

Spatial differentiation operations like curl, gradient and divergence require vector data in a

```
type :: vector_full
   real(kind(1.0D0)) :: covpsi
   real(kind(1.0D0)) :: covtheta
   real(kind(1.0D0)) :: covzeta
   real(kind(1.0D0)) :: conpsi
   real(kind(1.0D0)) :: contheta
   real(kind(1.0D0)) :: conzeta
   real(kind(1.0D0)) :: rad
   real(kind(1.0D0)) :: pol
   real(kind(1.0D0)) :: tor
   integer :: latest
end type vector_full

type(vector_full), dimension(nx,ny,nz) :: a
```

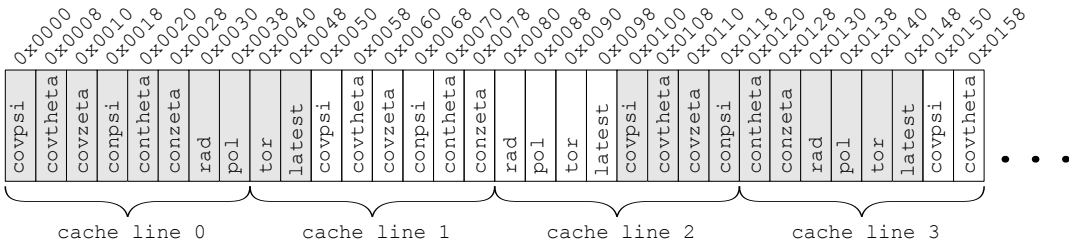


Figure 4.1: Illustration of the layout of an array of a ten element structure in memory and the placement of data on cache lines.

specific representation (in the case of curl and divergence) and/or output in a specific representation (curl and gradient). These operations will only touch one third of the data in each of the vector arrays they read or write if the other representations are not updated. However, this does not mean they will only load one third of the data from main memory. As all microprocessors use blocks or lines to divide the cache (see Section 2.3.1) each cache line will be over half "empty" when it is loaded into the cache.

Every compiler tested keeps derived data in a similar method to C "structs", as contiguous sections of memory. Arrays of derived types are mapped into memory as shown in Figure 4.1. Operations requiring data from only one of the three basis sets have to load data a cache line at a time, meaning in the majority of cases components from the other representations are loaded along with the required data as part of the cache line. This has the effect of polluting the cache with useless data, reducing its effective size to approximately $\frac{3}{10}$ of its actual size and causing three times as much data to be loaded from memory than necessary.

Figure 4.2 shows the results of running a sample curl operation on HECToR with the two different memory structures compiled using the PGI compiler. One uses an array of three component vectors, loading data in one representation and storing it in another, the other

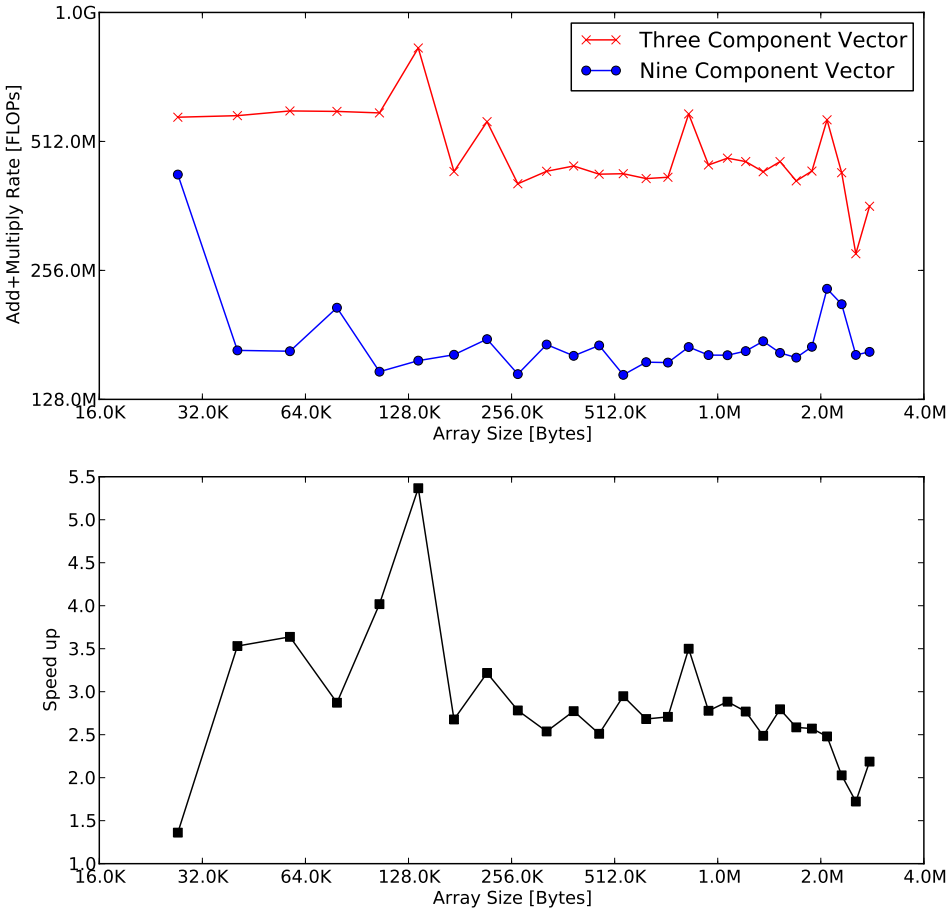


Figure 4.2: (Top) Load rates performing a curl operation on different data volumes using nine components and three components. (Bottom) speed up of three components over the nine components. Run on HECToR, compiled using PGI.

uses an array of nine component vectors, reading from only three components and writing to a different set of three. The upper graph shows the absolute values for the number of floating-point operations performed per second against the size of the input array and the lower graph shows the relative speed difference between the three and nine component methods.

The results show that there is a strong advantage to using only three components against nine components, even though the number of floating-point operations being performed (and the ordering) are the same. Having all the data stored contiguously in memory offers a significant speed boost (on average 2.83 times faster) on architectures which use caches (i.e. almost every modern architecture).

Converting CENTORI to only operate with one representation at a time and converting between the representations as and when required would improve CENTORI's performance significantly. The amount of data being read and written to memory would be reduced and all of the data loaded to cache would be referenced as all the data is useful at all times. The amount of calculation would also be reduced as the data would only be converted between representations when required, rather than pre-emptively in the current scheme.

4.2 Choosing a New Memory Layout for CENTORI

So, CENTORI should be changed to use data structures that store entire fields of data in only one of the three coordinate representations with a single integer for the entire field to track the status. Should a different representation be required, the components of the entire field should be recomputed and the integer updated. This should not incur any additional computational cost, indeed it may reduce the amount of computation. Currently every time one representation is updated, the other two are pre-emptively recalculated, whereas the proposed system will only convert the data when required. There may be some pathological cases where constant vectors are used in different formats and data has to be converted between formats more than once, however these are expected to be rare.

4.2.1 Designing a new format

So how should the new, more compact, format be structured in memory to achieve the best performance? Should the data be structured in the same way as the original format, as a derived type, or should the data be stored in a native array of Fortran doubles? If so, how

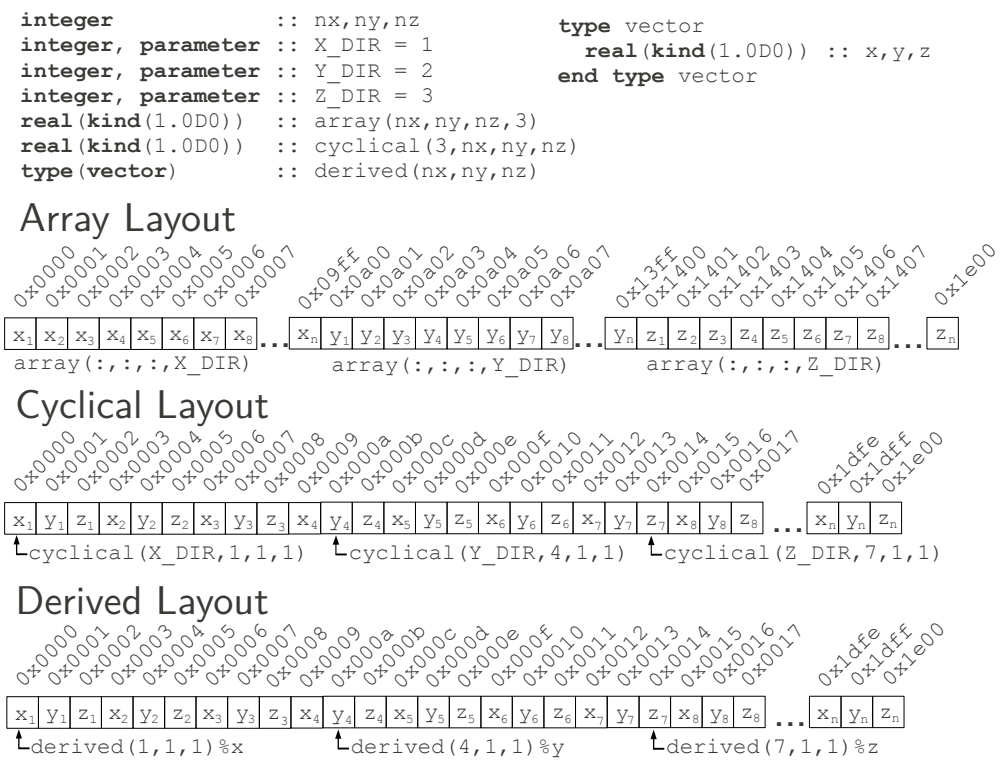


Figure 4.3: Proposed memory layout diagrams for a single representation format for CENTORI.

should this array be structured?

Three possible models are proposed and their layout in memory is represented in Figure 4.3. They are:

cyclical a four dimensional array of double precision floating-points with the vector component as the fastest changing dimension. Each component is contiguous with the other components for the same data point.

derived an array of a derived type which has three double precision floating-point values. On most compilers this will have the same structure in memory as the cyclical format, though is referenced in the language in a different way.

arrays a four dimensional array of double precision floating-points with the vector component as the slowest changing dimension. Each component is contiguous with the same component for all data points. This is similar to storing each component in a separate three dimensional array.

Some operations like addition, subtraction, scaling or division are not dependent on the

memory layout. Other operations are sensitive to the memory layout and their performance may change as a result. It is possible to offer arguments as to why each format is advantageous compared to the others, for example:

- The cyclical and derived formats will have the same structure in memory when compiled with most standard compilers. However, the cyclical format, being an array of native types, uses less sophisticated Fortran statements and so the compiler may be able to apply better optimisations.

- The cyclical format only requires one stream of data from memory for all the components. If data is accessed sequentially the subsequent components will be loaded into cache. In the array three separate streams to memory are required if each component is accessed at each data point in sequence.

- The array format allows finite difference operations used to calculate the curl or divergence to be vectorised. Figure 4.4 demonstrates how a loop calculating the difference between two x components cannot be vectorised when the data is cyclical format. The input results are not contiguous in memory. The array format does place the arguments so they are contiguous in memory allowing the individual subtract operations to be replaced with a single packed vector instruction.

- The cyclical format will poison the cache during a finite difference operation as shown in Figure 4.4 as the system will be forced to load all the components as they will be on the same cache line. The array format positions data for the same component on the same cache line and so all data loaded will be used.

As converting the whole of CENTORI to a new format requires rewriting large sections of the source code, identifying the best memory layout on HPCx, HECToR and HPC-FF before rewriting is a wise precaution. As it is difficult to identify which format will provide the best performance without any practical experience, a series of experimental benchmarks is proposed that test the performance of important kernels.

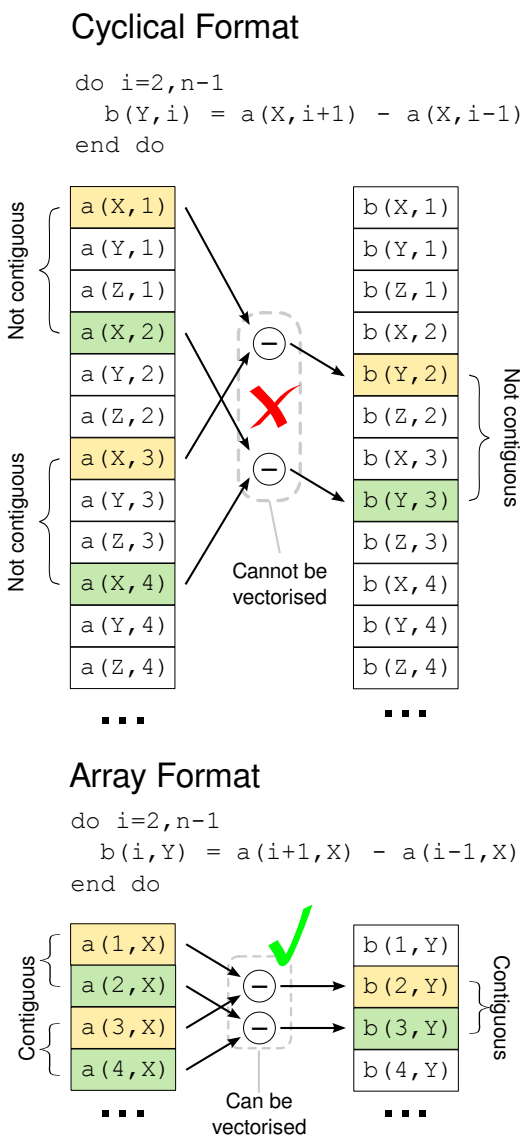


Figure 4.4: This figure illustrates the challenges of vectorising a simple loop over data in the cyclical format and the potential when using the array format.

Name	Mathematical Operation
add	$\mathbf{c} = \mathbf{a} + \mathbf{b}$
scale	$\mathbf{a} = \mathbf{a} * b$
curl	$\mathbf{b} = \nabla \times \mathbf{a}$
divergence	$b = \nabla.\mathbf{a}$
gradient	$\mathbf{b} = \nabla a$
cross	$\mathbf{c} = \mathbf{a} \times \mathbf{b}$
dot	$c = \mathbf{a}.\mathbf{b}$
L2 norm	$b = \mathbf{a}.\mathbf{a}$
*scaleadd**	$\mathbf{c} = k\mathbf{a} + l\mathbf{b}$
convert	(physical/covariant/contravariant)

*scaleadd is not originally directly used by CENTORI, however does represent a common operation which is expected to perform well on processors with separate add and multiply floating-point units.

Table 4.1: A list of the primary operations implemented as high performance kernels and their mathematical descriptions.

4.3 Benchmarking the Performance of the Memory Hierarchy

To identify which format performs best the tests should be run on every platform that CENTORI is expected to run on. The tests consist of a limited set of mathematical kernels that form a significant part of CENTORI's performance profile. The results of the tests will provide more information on the choice of memory layout.

Table 4.1 is a list of the kernels that profiling shows are significant contributors to the run time. These operations are relatively simple when expressed in mathematical notation, but vary in complexity when realised as source code. Operations like curl, divergence and gradient are the most complicated, while operations like add and scale are the simplest. Some operations have been chosen to represent whole classes of similar operations, for example $\mathbf{c} = \mathbf{a} + \mathbf{b}$ and $\mathbf{c} = \mathbf{a} \star \mathbf{b}$ are expected to perform identically on all the target architectures so only one is used in the benchmark.

Hand optimised kernels

A simple test would be to compare each of the formats with kernels written in straight forward Fortran without any additional optimisations that target the architecture or the memory format. This would test the ability of the compiler to optimise the source code for the memory format on each architecture. However the ultimate goal is to identify the memory format and kernel that performs best overall. So by generating a suite of kernels, which include hand tuned optimisations, the result is not solely dependent upon the compiler's ability to optimise simple

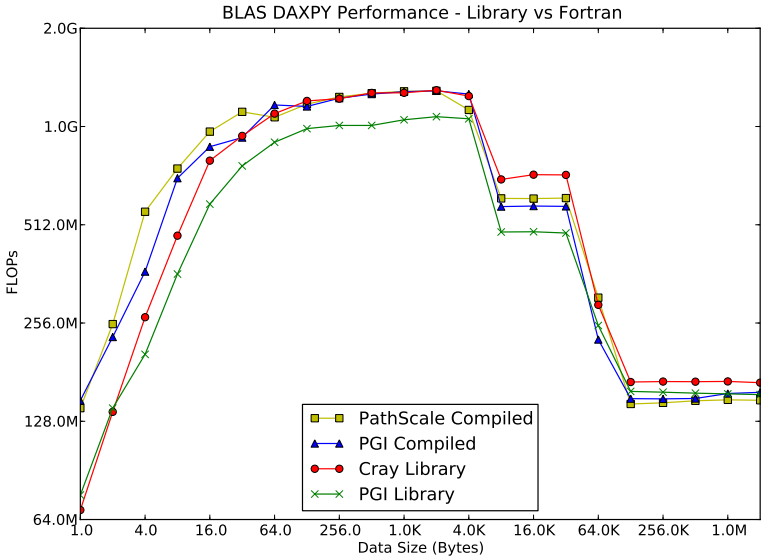


Figure 4.5: Comparing the performance of the DAXPY routine from bundled Cray and PGI BLAS libraries with the equivalent operation written and compiled from Fortran source with PGI and Path. Tested on HECToR's Phase 1 Opteron 290 2.8 GHz Dual Core processors.

code but of a human programmer's ability to identify and implement relevant optimisations. This biases the benchmarks towards the memory formats that are more readily optimised by hand, but this is an important aspect in assessing the performance of the layout.

Using external libraries

The quickest solution might be to use an existing high performance library which provides the functionality required by CENTORI on the target architectures. The Basic Linear Algebra Subprograms (BLAS)[75] suite is an application programming interface for a library of subroutines relating to common linear algebra operations, some of which are found in CENTORI. Hardware vendors publish highly optimised implementations of each subroutine for specific architectures which provide near optimal performance to any application using the subroutines. The advantage to the vendors is in only having to write one version of the library to optimise all codes which use the library while the developer can be confident of good performance on a wide variety of systems. Applications become easier to maintain because the algorithmic logic is separated from hardware specific optimisation work by the BLAS interface. Using BLAS will generally provide acceptable performance levels on all platforms where a tuned library exists, providing "portable" optimisation.

Complications arise as many of the more sophisticated operations in CENTORI cannot be easily expressed as BLAS operations. Large parts of the code would still require hand tuned subroutines though others could use the optimised BLAS routines. An example of a code that could be replaced by BLAS is the "scaleadd" kernel which matches the Basic Linear Algebra System (BLAS) Level 1 routine DAXPY. DAXPY computes $\mathbf{A} = \alpha\mathbf{B} + \mathbf{A}$ and it is useful to compare the performance of the library against equivalent compiled Fortran source code. Figure 4.5 plots the result on a Dual Core Opteron Santa Ana (from an earlier phase of HECToR, Phase 1) operating at 2.8GHz. These results show that the compiler is capable of generating executable code that has equivalent or better performance than the optimised BLAS library when the arrays are held in cache.

This result demonstrates that for a code like CENTORI, which can only partially make use of BLAS, it is preferable to have subroutines which perform the exact algorithm required and achieve a good level of performance rather than try to adapt CENTORI to fit into BLAS's function calls and realise only a small performance gain.

4.4 The Benchmarks

Each operation in Table 4.1 has a basic source code implementation for each of the three memory layouts. These baseline or "naive" kernels are the canonical implementations of the algorithm for each operation and layout combination, examples of these kernels for the "derived" layout are shown in Appendix A.7. Hand optimised versions of the kernels have also been developed which are compared against the results for the canonical "naive" kernels for validation.

There are many optimisation techniques that can be applied to the kernels. Some of the most important hand optimisations include:

- *explicitly unrolling loops* - this involves manually repeating the contents of a loop a set number of times to reduce the number of iterations through the loop. This is a very common compiler optimisation and it is unlikely that manual unrolling would out-perform the compiler.

- *fusing loops* - some algorithms contain loops which operate over the same range. By combining the contents of the two loops into a single loop (if possible) it may improve the overall performance of the code by allowing greater hardware parallelism or reducing memory traffic.

Example 4.1 A loop which calculates a centred finite partial difference on a scalar three dimensional array in any direction depending on the value of `sep`.

```
od = 1.0 / dx
do i = 1+sep,n-sep
  out(i) = (a(i+sep) - a(i-sep)) * od
end do
```

– *splitting loops* - some loops contain very complicated memory access patterns which may "confuse" the compiler or overload hardware like memory pre-fetch engines. Splitting the loop into two less complicated loops may improve the performance overall.

– *reshaping arrays* - arrays in Fortran are defined to have a row-major layout in memory, operations on arrays with three dimensions usually require three nested loops, however passing the array into a subroutine which maps the problem to a one dimensional array can reduce the number of loops to only one.

– *finite difference in any direction* - when using the array memory layout format the finite difference operation, which forms the basis of the curl, divergence and grad operations, can be expressed as a one dimensional loop as in Example 4.1. The direction of the derivative (e.g. $\frac{d}{dx}$, $\frac{d}{dy}$, $\frac{d}{dz}$) is determined by value of `sep`. for $\frac{d}{dx}$ then `sep=1`, for $\frac{d}{dy}$ then `sep=nx` and for $\frac{d}{dz}$ then `sep=nx*ny`.

A typical example of a canonical kernel is Example 4.2, which is an implementation of the "add" kernel for the cyclical memory layout. This version uses three nested loops to iterate over each dimension of the arrays. Due to Fortran's row-major indexing the values in each array are accessed sequentially and a significant proportion of the data will be preloaded into cache as part of a cache line and through explicit hardware pre-fetching if supported.

An alternative method of writing the same code is to use Fortran's requirement that arrays are all contiguous blocks of memory in row major format. This allows the four dimensional array to be reshaped to an equivalently sized 1D array and processed as a single loop as shown in Example 4.3. A further method would be to use Fortran array notation to express the operation as in Example 4.4.

Each method performs the same operation and in the case of Examples 4.2 and 4.3, in the same order. Example 4.2 removes two levels of loop overhead by using a single loop and so should execute more quickly. Ideally an optimising compiler would recognise the equivalence

Example 4.2 Canonical implementation of the add kernel for the cyclical memory layout.

```
integer, parameter :: X=1,Y=2,Z=3
real(kind(1.0D0)), dimension(3,nx,ny,nz) :: a,b
real(kind(1.0D0)), dimension(3,nx,ny,nz) :: c

do k=1,nz
  do j=1,ny
    do i=1,nx
      c(X,i,j,k) = a(X,i,j,k) + b(X,i,j,k)
      c(Y,i,j,k) = a(Y,i,j,k) + b(Y,i,j,k)
      c(Z,i,j,k) = a(Z,i,j,k) + b(Z,i,j,k)
    end do
  end do
end do
```

Example 4.3 An optimised version of the add kernel in Example 4.2 which reshapes the arrays to 1 dimension.

```
real(kind(1.0D0)), dimension(3*nx*ny*nz) :: a,b
real(kind(1.0D0)), dimension(3*nx*ny*nz) :: c

do i=1,3*nx*ny*nz
  c(i) = a(i) + b(i)
end do
```

Example 4.4 The same expression as Examples 4.2 and4.3 in Fortran array notation.

```
real(kind(1.0D0)), dimension(3,nx,ny,nz) :: a,b
real(kind(1.0D0)), dimension(3,nx,ny,nz) :: c

c = a + b
```

Kernel	array	cyclical	derived
add	14	4	3
scale	7	8	2
curl	6	3	4
divergence	3	3	3
gradient	6	3	4
cross	6	4	3
dot	2	2	2
L2 norm	3	2	2
scaleadd	12	11	5
convert	9	9	7
Total	68	49	35

Table 4.2: Number of different kernel implementations for each operation and memory layout (total 152).

Property	ECDF	HECToR	BlueGene/L	HPCx
Manufacturer	Intel	AMD	IBM	IBM
Micro architecture	Core	Barcelona	Power 440	Power 5+
Architecture	x86	x86	PowerPC	PowerPC
Cores per module	4	4	2	4
Clock Speed	3000MHz	2300MHz	700MHz	1500 MHz
FLOPS per clock	4	4	4	4
SIMD Instructions	SSE1-4	SSE1-4	DFPU	None

Table 4.3: Comparison of some of the features of the modern processors currently under investigation.

of all the examples and replace them with the most optimised versions, however with many compilers this is not the case.

A large number of hand optimised kernels have been created for each memory layout. Table 4.2 lists the number of versions of each kernel and memory layout. A point of note is that much of the early optimisation work was done on the array memory layout which generated a large number of different kernels. The experience gained was applied to the optimisation of kernels for the cyclical and derived memory layouts and many of the poor optimisations tested in array format were not ported across to the cyclical and derived memory layouts. This explains the larger number of optimised kernels for the array format than for the cyclical and derived formats.

4.4.1 Compilers and Platforms

The test suite was compiled for the four platforms listed in Table 4.3 and for each of the three compilers on HECToR. Table 4.4 documents each compiler and option used in the test suite.

ECDF[76] is a Linux based cluster operated by the University of Edinburgh. The micropro-

Platform	Compiler Vendor	Version	Options
BlueGene/L	IBM	10.1	`-O3 -qhot -qtune=440 -qarch=440d` *
HPCx	IBM	10.1	`-O3 -q64`
HECToR	PGI	8.0-2	`-fastsse -O3`
HECToR	PathScale	3.1	`-O3 -OPT:Ofast`
HECToR	Cray	7.0.4	`-O3 -r a`
ECDF	Intel	10.1	`-O3 -heap-arrays 8` ‡

* This option allows the IBM compiler to include SIMD instructions for use on BlueGene.

‡ This option forces the compiler to place automatic arrays over 8 KB onto the heap rather than the stack

Table 4.4: List of compilers and option combinations used to create the benchmark.

Exclusive Field Size	Including Halos	Vector field volume
$8 \times 8 \times 8$	$10 \times 10 \times 10$	23 KB
$16 \times 16 \times 16$	$18 \times 18 \times 18$	137 KB
$8 \times 8 \times 1$	$10 \times 10 \times 3$	2.3 KB
$16 \times 16 \times 1$	$18 \times 18 \times 3$	7.6 KB
$32 \times 32 \times 1$	$34 \times 34 \times 3$	27.1 KB

Table 4.5: Table showing the volume of different sizes of vector fields in memory.

cessor architecture is similar to that of HPC-FF, which was unavailable at the time that these experiments were performed. ECDF was used as a model for performance on HPC-FF before it became available. They differ only slightly in clock speed, HPC-FF is 2.93GHz to ECDF's 3.00GHz, but there are some fundamental differences in the memory architecture.

Each set of compiler options are the "standard" recommended instructions by the operators and vendors of the target systems. These generally include high optimisation through "-O3" and options to enable vector instructions.

4.4.2 Problem Sizes

Each kernel is capable of running on arrays of almost any size, however the test is only run on three array sizes which are specifically of interest to CENTORI. A standard model size for CENTORI is $128 \times 128 \times 128$ and only cubic processor decompositions are considered. Each processors works on a smaller subset of the global data set which will include the halo data from neighbouring processors.

Table 4.5 lists common cubic decompositions and the associated data volumes on the local processors. The kernels are tested with the equivalent arrays of size $18 \times 18 \times 18$ and $10 \times 10 \times 10$ to represent the appropriate decompositions. Due to subsequent work in Chapter 7 the tests were also run on arrays of size $34 \times 34 \times 3$, $18 \times 18 \times 3$ and $10 \times 10 \times 3$, which represent "slices" in the ζ direction. These are required to implement cache optimisation routines that are covered

in subsequent chapters.

4.4.3 Validating the results

Naturally, the hand optimised kernels must produce the correct numerical results. To check this each hand optimised version of the kernel is verified against the canonical version. Both versions are passed the same (random) data as input and the average relative difference of the outputs is measured over all points outside of the halo regions. Kernels are deemed to produce the same results if the average relative difference between points is below 10^{-10}.

Data inside the halos is not considered during the validation to allow for optimisations that "destroy" halo data in finite difference operations like curl, divergence and gradient. This allows the optimisation to place any data they require (or none at all) in the halo without failing the validation. Operations that do not destroy halo data, like add, scale, cross, are expected to calculate the correct results for all data points, including halos, correctly.

4.4.4 Running the tests

The test suite calculates the size of the clock tick of the high resolution timer, `MPI_Wtime`. Each test is then run repeatedly until an appropriate number of clock "ticks" has passed ($> 10^5$) which is determined from an initial estimate. The overall test sequence for each kernel is:

1. Generate input fields as a field of random number, between acceptable bounds (± 50.0). Reset the output fields to zero.
2. Time the kernel to approximate the result, this also loads the input data into the caches.
3. Calculate the number of repetitions required for sufficient ticks of the clock to pass.
4. Run the kernel for the required number of steps and calculate the average time per call.
5. Repeat step 4 multiple times (10) and calculate the average and variance of the results.
6. Compare the results against canonical version of kernel and record boolean result (verified).

These tests do not make any attempt to simulate realistic cache configurations. As each test is repeated multiple times, if all data can fit on cache then it will remain in cache, if all the data cannot be held on cache simultaneously then it will be stored on a lower memory level[77].

4.4.5 Compound kernels and temporary data fields

Testing the kernels in Table 4.1 as individuals does not accurately represent how the operations are performed in CENTORI. Though the discrete performance of each kernel provides valuable information, repeatedly calling the same kernel is not realistic. The input and output data will always be at the highest cache level whereas in CENTORI the input data may be any in part of the memory hierarchy, depending on what operations have been called previously.

A further influence on the performance is the generation of temporary data. All of the kernels tested operate on a maximum of two operands, so operations which are any more complicated will generate temporary arrays of data which form the input to other operations. This temporary data has to be stored somewhere and will occupy space in the cache and may be the cause for other data to be evicted. The interaction of the various levels of the memory hierarchy has a significant influence on the overall performance of CENTORI which calling single kernels does not model.

An alternative method of testing the kernels is to perform a sequence of operations from CENTORI. This method is a better simulation of the interaction between kernels and provides more information about the realistic performance that can be expected from a memory layout.

Two complex kernels have been extracted from CENTORI which represent a variety of simple kernels, these are:

Faraday Equation

This performs the operations used to evaluate the right hand side of the Faraday equations that were extant in June 2008 in CENTORI. The operation performed is:

$$\mathbf{X} = \nabla \times ((\mathbf{A} \times \mathbf{B}) + (\mathbf{C} \times \mathbf{B}) + p\mathbf{C} + q\nabla L) \tag{4.1}$$

where $\mathbf{A}$,$\mathbf{B}$ and $\mathbf{C}$ are vector fields, L is a scalar field and p and q are scalar constants.

Source Term

This evaluates the source term for the momentum equation which incorporates some of the simple kernels not included in the Faraday Equation. The operation performed is:

$$\mathbf{X} = (\mathbf{A} \times \mathbf{B}) + p\nabla(L + M) + \nabla(\frac{1}{2}\mathbf{C}.\mathbf{C}) \tag{4.2}$$

where **A**,**B** and **C** are vector fields, L and M are scalar fields and p is a scalar constant.

Each compound kernel calls multiple simple kernels, so if there are multiple hand optimised versions of each of the simple kernels which ones should be used? Each compound kernel has a version constructed from the canonical versions of the simple kernels and is used as the baseline for verification and performance.

The results of benchmarking the simple kernels are then used to generate the optimised versions of the compound kernels. The fastest versions of each kernel for each problem size on each architecture and compiler are selected and used to generate a new optimised version of the compound kernel for that architecture. These are the hand optimised compound kernel versions.

4.5 Results

The benchmarks were designed to answer one question: "Which memory layout will achieve the fastest performance in CENTORI?". Each test result only provides information about a single operation for a single memory layout, so a metric is required that combines the results from many individual kernels into a single score. The simplest method would be to take the average performance over the ten kernels and use this, however it does not reflect the composition of CENTORI's time step where some kernels are called more often than others.

4.5.1 Complex Kernels

The Faraday and Source kernels are small sections of CENTORI's source code. They are constructed by taking the fastest performing kernels for each platform and combining them to form the result. In CENTORI itself, both the Faraday and Source kernels are called once per time step so the total execution time for both is a small model of how a full CENTORI time step will perform. Table 4.6 records the amount of time to execute the Faraday and Source kernels in nanoseconds divided by the number of grid points in the problem. This metric allows comparison between problem sizes and platforms, figures in bold represent the smallest values for any platform and problem size combination.

The results show that the smaller problem sizes are faster per grid point than the larger problems. This is a function of two things:

– The operations curl and gradient, which are part of the complex kernel, require halo

Platform	Compiler	Layout	$10 \times 10 \times 3$	$18 \times 18 \times 3$	$34 \times 34 \times 3$	$10 \times 10 \times 10$	$18 \times 18 \times 18$
		array	693	753	672	919	822
BlueGene/L	*IBM*	cyclical	**398**	**459**	**453**	**489**	**493**
		derived	422	484	496	506	530
		array	**50**	**58**	**56**	**66**	**68**
ECDF	*Intel*	cyclical	53	60	61	**66**	69
		derived	52	60	64	**66**	69
		array	**53**	**51**	**88**	**59**	**175**
HECToR	*PGI*	cyclical	54	52	91	60	183
		derived	54	72	98	82	185
		array	59	74	81	113	141
HECToR	*Cray*	cyclical	**42**	**56**	**71**	**62**	114
		derived	45	58	73	64	**113**
		array	64	74	84	86	133
HECToR	*PathScale*	cyclical	52	66	**78**	74	128
		derived	**48**	**62**	**78**	**71**	**125**
		array	116	103	98	137	130
HPCx	*IBM*	cyclical	90	84	83	102	111
		derived	**84**	**81**	**82**	**98**	**110**

Table 4.6: The measured cost of the combined Faraday+Source kernels per data grid point [nanoseconds]. Values in bold are the fastest for a particular architecture and problem size.

data to operate, so will only operate on 1 point along the z coordinate in the $N \times N \times 3$ problem sizes, as opposed to $N - 2$ points in the z coordinate in the $N \times N \times N$ problem sizes. Proportionally these kernels will perform fewer calculations on smaller problems than larger.

– Larger problems will exhaust the caches in the memory hierarchy and have to read and write data from and to the slower main memory. HECToR's Opterons have a smaller volume of cache and so demonstrate a large overall increase in the per grid point time between $10 \times 10 \times 10$ and $18 \times 18 \times 18$ than other platforms.

The figures show that HECToR has the best performance on these kernels on the fastest problem size, $10 \times 10 \times 3$, being 1.2 times faster than ECDF, 2 times faster than HPCx and 9.5 times faster than BlueGene/L. This is despite ECDF having a much larger cache and higher clock speed than HECToR.

Table 4.7 summarises the previous table and shows the fastest memory layout for each platform and problem size. The results show that there are single memory layouts which perform well on individual platforms, however there is not a single memory layout that is optimal on all platforms. Taking the three target architectures, ECDF favours "array", HPCx "derived" and each of the different compilers on HECToR favours a different layout. Each problem size has a fastest compiler which makes choosing a preferred compiler from each of the

Platform	Compiler	$10 \times 10 \times 3$	$18 \times 18 \times 3$	$34 \times 34 \times 3$	$10 \times 10 \times 10$	$18 \times 18 \times 18$
BlueGene/L	*IBM*	cyclical	cyclical	cyclical	cyclical	cyclical
ECDF	*Intel*	array	array	array	array	array
HECToR	*PGI*	array	array	array	array	array
HECToR	*Cray*	cyclical	cyclical	cyclical	cyclical	derived
HECToR	*PathScale*	derived	derived	derived	derived	derived
HPCx	*IBM*	derived	derived	derived	derived	derived

Table 4.7: The fastest memory layout for each processor/compiler combination and problem size tested, using Faraday and Source measurements.

three difficult.

The data show that there is no layout which performs optimally on all platforms so instead, what are the consequences of choosing the wrong layout? Table 4.8 shows the percentage increase in cost of the Faraday+Source kernels associated with memory layouts which are not optimal.

- On all platforms, there is only a relatively small decrease in performance associated with switching to cyclical where it does not already have the best performance, on average 3.5%.

- There is a much greater cost to choosing the derived or array layouts if either is not optimal, 8.2% and 38.4% respectively on average.

Therefore these results would suggest that using cyclical layout would provide the best universal performance on most platforms. This is because the platforms which favour the array layout, ECDF and HECToR - PGI, do so only marginally, whereas those platforms which favour cyclical do so strongly.

However, the Faraday and Source kernels are not complete models for CENTORI. There are many operations which are called many times in the main code that are not called once in either kernel. Table 4.9 show the ratio of the calls made in CENTORI, whereas Table 4.10 shows the number of calls made by each operation in a single time step in CENTORI. The top two calls by frequency in the main CENTORI are not present in the Faraday or Source kernels and so these may not present a comprehensive estimate of CENTORI's performance.

4.5.2 Weighted averages

In an attempt to create a more accurate representation of the composition of calls in CENTORI a second metric is defined for each platform, problem size and memory layout. The metric is

Platform	Compiler	Layout	$10 \times 10 \times 3$	$18 \times 18 \times 3$	$34 \times 34 \times 3$	$10 \times 10 \times 10$	$18 \times 18 \times 18$
		array	74%	64%	49%	88%	67%
BlueGene/L	*IBM*	cyclical	-	-	-	-	-
		derived	6%	6%	10%	3%	7%
		array	-	-	-	-	-
ECDF	*Intel*	cyclical	5%	3%	9%	1%	1%
		derived	2%	3%	14%	0%	1%
		array	-	-	-	-	-
HECToR	*PGI*	cyclical	1%	2%	3%	2%	4%
		derived	1%	40%	12%	39%	6%
		array	41%	32%	14%	82%	25%
HECToR	*Cray*	cyclical	-	-	-	-	1%
		derived	6%	3%	2%	3%	-
		array	33%	20%	8%	22%	7%
HECToR	*PathScale*	cyclical	8%	6%	-	5%	3%
		derived	-	-	0%	-	-
		array	37%	26%	20%	40%	18%
HPCx	*IBM*	cyclical	7%	3%	1%	4%	1%
		derived	-	-	-	-	-

Table 4.8: Percent additional cost over the fastest memory layout when choosing an alternative, from Faraday and Source measurements.

Benchmark	Calls (n_i)
Add	5 (29%)
Scale	4 (24%)
Cross Product	3 (18%)
Gradient	3 (18%)
Curl	1 (6%)
L2 Norm	1 (6%)
Total	17

Table 4.9: Number of calls to each kernel in the Faraday+Source Kernels.

Benchmark	Calls (n_i)
Convert	75 (43%)
L2 Norm	28 (16%)
Curl	24 (14%)
Gradient	12 (7%)
Addition	10 (6%)
Divergence	9 (5%)
Dot product	9 (5%)
Cross product	6 (3%)
Total	175

Table 4.10: Number of times each operation is called in a single CENTORI time step.

Platform	Compiler	Layout	$10 \times 10 \times 3$	$18 \times 18 \times 3$	$34 \times 34 \times 3$	$10 \times 10 \times 10$	$18 \times 18 \times 18$
		array	**1141**	2786	3694	3424	4626
BlueGene/L	*IBM*	cyclical	1234	**2609**	**3286**	**3139**	**3976**
		derived	1894	3755	4367	4231	5065
		array	**254**	**469**	**503**	**545**	**624**
ECDF	*Intel*	cyclical	305	521	543	611	663
		derived	309	523	548	609	668
		array	**312**	**377**	**530**	**426**	**683**
HECToR	*PGI*	cyclical	344	396	543	452	719
		derived	414	453	552	557	746
		array	**300**	**310**	**517**	**395**	**643**
HECToR	*Cray*	cyclical	335	348	531	426	653
		derived	374	392	544	477	659
		array	429	456	594	570	755
HECToR	*PathScale*	cyclical	**402**	**429**	**582**	**535**	**732**
		derived	407	438	633	546	781
		array	577	668	684	849	916
HPCx	*IBM*	cyclical	531	641	684	815	936
		derived	**523**	**629**	**668**	**791**	**908**

Table 4.11: Estimated "cost" per grid point of a time step using a weighted sum by call frequencies in CENTORI (nanoseconds).

the sum, c, of the product of the frequency of calls, n_i, and the fastest performing version of the kernel, t_i.

$$c = \sum_{i \in \{\text{kernels}\}} t_i n_i$$

The value c is an estimate of the run time of the kernel that has the same number of calls as a time step in CENTORI. The frequency of calls, n_i, is recorded in Table 4.10. This method is a more accurate reflection of the combination of operations used in CENTORI than the Faraday+Source kernel, however it does have its own weaknesses:

- Firstly the number of calls has been determined from the name of subroutine calls in CENTORI. There are a number of disenfranchised operations in the "*Miscellaneous Evolution*" that cannot be seen in the sampling and so are not counted or represented.
- Secondly, the timings of the kernels are in idealised conditions. They do not represent the wide variety of conditions that the kernel will be run in (i.e. the state of the data in cache) very accurately and take no account of the effect interactions between operations will have on the performance.

Table 4.11 records the "cost" per grid point for each memory layout when performing a time step in CENTORI for each architecture and problem size.

Platform	Compiler	$10 \times 10 \times 3$	$18 \times 18 \times 3$	$34 \times 34 \times 3$	$10 \times 10 \times 10$	$18 \times 18 \times 18$
BlueGene/L	*IBM*	array	cyclical	cyclical	cyclical	cyclical
ECDF	*Intel*	array	array	array	array	array
HECToR	*PGI*	array	array	array	array	array
HECToR	*Cray*	array	array	array	array	array
HECToR	*PathScale*	cyclical	cyclical	cyclical	cyclical	cyclical
HPCx	*IBM*	derived	derived	derived	derived	derived

Table 4.12: The fastest memory layout for each processor/compiler combination and problem size tested, using weighted sum estimates.

- The overall predicted cost of a time step is 5-6 times larger for the $10 \times 10 \times 3$ problem size than the results for the Faraday+Source kernels. The overall number of operations increased by ten times, implying that the time step average is made from a greater proportion of faster operations than the Faraday+Source kernels.

- The kernels are run in isolation, so the significant decrease in performance when operating on a larger problem, that was seen in the Faraday+Source kernels on HECToR, is not as severe with this metric. e.g. a two fold decrease in performance compared to a three fold on the earlier metric.

The results again show that there is no single memory layout that performs best across architectures and memory sizes, though each architecture does perform best using a single layout on most problem sizes. On HECToR, where there are three compilers, the fastest compilers, Cray and PGI, agree on "array" as the fastest format. Table 4.12 summarises the results.

Table 4.13 performs the same comparison as Table 4.8 for the weighted sum metric. The average cost in performance in switching from the fastest kernel to array is 6.8%, to cyclical 6.7% and to derived 18.5%, effectively discounting derived as the choice of layout, but leaving array or cyclical.

4.5.3 Qualifying error in the weightings

The weightings sum's major weakness is that it does not reflect the interaction of kernels in CENTORI as well as the Faraday+Source kernel does. However it is possible to estimate this error by using the frequency of operations in the Faraday+Source kernels to estimate the performance of the Faraday+Source kernel. Table 4.14 shows the result of this comparison for HECToR using the Cray Compiler.

The results show several trends:

Platform	Compiler	Layout	$10 \times 10 \times 3$	$18 \times 18 \times 3$	$34 \times 34 \times 3$	$10 \times 10 \times 10$	$18 \times 18 \times 18$
		array	-	7%	12%	9%	16%
BlueGene/L	*IBM*	cyclical	8%	-	-	-	-
		derived	66%	44%	33%	35%	27%
		array	-	-	-	-	-
ECDF	*Intel*	cyclical	20%	11%	8%	12%	6%
		derived	21%	12%	9%	12%	7%
		array	-	-	-	-	-
HECToR	*PGI*	cyclical	10%	5%	2%	6%	5%
		derived	33%	20%	4%	31%	9%
		array	-	-	-	-	-
HECToR	*Cray*	cyclical	12%	12%	3%	8%	2%
		derived	25%	26%	5%	21%	2%
		array	7%	6%	2%	7%	3%
HECToR	*PathScale*	cyclical	-	-	-	-	-
		derived	1%	2%	9%	2%	7%
		array	10%	6%	2%	7%	1%
HPCx	*IBM*	cyclical	2%	2%	2%	3%	3%
		derived	-	-	-	-	-

Table 4.13: Percent additional cost over the fastest memory layout when choosing an alternative, from weighted sum estimates.

	Layout	$10 \times 10 \times 3$	$18 \times 18 \times 3$	$34 \times 34 \times 3$	$10 \times 10 \times 10$	$18 \times 18 \times 18$
	array	17.7	71.5	281.6	112.8	824.8
Measured	cyclical	12.6	54.1	247.7	62.0	664.4
	derived	13.4	55.9	252.0	64.0	660.5
	array	9.4	31.5	162.1	39.8	328.0
Estimated	cyclical	10.5	35.2	165.3	42.5	336.1
	derived	12.2	41.2	179.7	48.2	360.9
	array	88%	127%	74%	183%	151%
Difference	cyclical	19%	54%	50%	46%	98%
	derived	10%	36%	40%	33%	83%

Table 4.14: Comparison of the estimated time to complete the Faraday and Source Kernels versus the measured values on HECToR using the Cray compiler.

- The estimate is consistently for better performance than observed in the real Faraday+Source. This is not unexpected for reasons explained previously.

- The difference in performance increases as the problem size increases, indicating that the increased performance is a function of the increasing data volumes.

- The difference in performance is much greater for the array kernels (averaging 125%) than the cyclical (average 53%) or derived (40%).

4.6 Conclusions from the Benchmark

The benchmarks have provided the following information:

- No memory layout has been shown to provide optimal performance on all platforms with either metric. Therefore it is best to find a "*least worst layout*"

- The Faraday+Source kernels, which represent the interaction between kernels more accurately indicate cyclical as the least worst layout.

- The Faraday+Source metric is not completely representative of all the operations performed in CENTORI and so the weighted average metric is a more accurate represention of the frequency of calls in CENTORI.

- The average cost in performance of choosing the wrong layout is 6.8% for array and 6.7% for cyclical according to this weighted average metric.

- The fastest compilers on HECToR and ECDF both indicated using array. These are the most important architectures for CENTORI as they are the newest and represent systems which already have a time budget for CENTORI.

- There is a higher difference in improved performance between the predicted weighted sum and the actual Faraday+Source kernels for the array layout than the cyclical or derived layouts.

- The Power architectures (HPCx and BlueGene/L) perform best using derived or cyclical memory layouts and the x86 architectures (HECToR and ECDF) perform best using the array layout.

– On many architectures there is a difference in the performance of the derived and cyclical layouts despite their similarity on first approximation. This is most noticeable with the BlueGene/L and on HECToR with the PGI and Cray compilers. This suggests the compilers are treating the code differently even though the memory layout is likely to be similar.

So in conclusion, there is no clear single layout that provides the best performance in CENTORI, though there are some reasons to use the array kernel on HECToR and ECDF. It is clear that different microprocessor architectures favour different memory layouts, and as CENTORI is likely to run on a variety of architectures during its operational lifetime hard-coding CENTORI to a single layout may involving re-engineering CENTORI in the future. Instead it would be better to adapt CENTORI to use any memory layout allowing it to select the best for the target architecture at compile time.

Chapter 5

A Domain Specific Library for CENTORI

Chapter 4 concludes that there is no single memory layout that can produce optimal performance across all the architectures CENTORI is expected to run on. With HPC services such as HECToR and HPCx undergoing regular upgrades throughout their life and computational resources being supplied by a variety of agencies, CENTORI must be flexible enough to adapt to these changing architectures. To perform optimally on any platform, it appears that CENTORI must have its memory layout tailored to the target architecture.

Changing CENTORI's memory layout requires rewriting large parts of the source code because CENTORI mixes high level concepts, like the choice of algorithm, with low level implementation detail like memory layout and parallel communication. Altering the memory layout affects code at every level of the program, from the ordering of the individual floating-point instructions in the tridiagonal solvers to the implementation of high level algorithms like the evaluation of the Momentum Equation. Example 5.1 shows an extract of CENTORI's source code that implements the evaluation of Equation 5.1.

$$\mathbf{B} = \mathbf{v}_i^{old} + \frac{v_A \Delta t}{2}(\mathbf{G}^* \times \mathbf{v}_i^{\text{old}}) + v_A \Delta t R_0 \mathcal{J}_{i,j} D_{v,j}(\mathbf{v}_{i,j+1,k}^{\text{latest}} + \mathbf{v}_{i,j-1,k}^{\text{latest}}) \tag{5.1}$$

Example 5.1 concatenates all of the operations in Equation 5.1 into a single set of nested loops which increases its computational intensity (the ratio of calculation to memory accesses) but makes the code hard to understand or extend. To maintain a separate version for each of the

Example 5.1 A section of source code from CENTORI performing the operation defined in Equation 5.1.

```
do k=1,nzeta
  do j=1,ntheta
    do i=1,npsi
      brhs(1,i,j,k) = oldvi(i,j,k)%rad + vadto2 &
          * ( gstar(i,j,k)%pol*oldvi(i,j,k)%tor &
          - gstar(i,j,k)%tor*oldvi(i,j,k)%pol ) &
          + vadt*raxis*local_jac(i,j)*odt2*dmid &
          * (vilatest(i,j+1,k)%rad              &
          + vilatest(i,j-1,k)%rad)              &

      brhs(2,i,j,k) = oldvi(i,j,k)%pol + vadto2 &
          * ( gstar(i,j,k)%tor*oldvi(i,j,k)%rad &
          - gstar(i,j,k)%rad*oldvi(i,j,k)%tor)  &
          + vadt*raxis*local_jac(i,j)*odt2*dmid &
          * ( vilatest(i,j+1,k)%pol             &
          + vilatest(i,j-1,k)%pol )             &

      brhs(3,i,j,k) = oldvi(i,j,k)%tor + vadto2 &
          * ( gstar(i,j,k)%rad*oldvi(i,j,k)%pol &
          - gstar(i,j,k)%pol*oldvi(i,j,k)%rad)  &
          + vadt*raxis*local_jac(i,j)*odt2*dmid &
          * ( vilatest(i,j+1,k)%tor             &
          + vilatest(i,j-1,k)%tor )             &
    end do
  end do
end do
```

memory layouts means duplicating (or in fact triplicating) the effort of developing, maintaining and testing CENTORI across different platforms.

A more sustainable approach is required to create an adaptable and flexible version of CENTORI that does not need a radical overhaul every time it switches platform. Disentangling CENTORI's bespoke high level algorithms from its generic low level functions would allow them to be optimised on each architecture without requiring an understanding of the high level functionality and vice-versa.

5.1 Abstraction and modularisation

Separating CENTORI into two parts, the high level algorithms and the low level implementation, requires an interface for the two to interact. By hiding complexity and detail, high level design decisions are separated from the detail of the implementation, allowing both to be changed without affecting the other. Using an abstraction like this is common across computer science and has several advantages:

- *Portability* — designs can be expressed in an abstract way, sufficiently generic that they can be executed by any low level implementation which fulfils the requirements. For example, a programming language allows high level concepts to be expressed in source code which can be translated/compiled into instructions that run on many different processors. This is the aim for CENTORI, to create multiple low level implementations for CENTORI's fundamental algorithms that perform optimally on any individual architecture.

- *Simplification and standardisation* – abstractions generally reduce the level of complexity by hiding the detail and providing a mechanism to express complicated concepts more succinctly e.g. high level programming languages provide simple ways to perform loops which are implemented using comparisons, increments and jump instructions.
  Designing a good abstraction for CENTORI would improve the readability of high level concepts and improve developer productivity.

Abstractions do incur some costs and penalties when they are used:

- *Inefficiency* — typically the more general an abstraction is, or if many layers of abstraction are used, the further the concepts are from the underlying hardware. Each concept has to be translated into smaller concepts in the level below which is more flexible but usually less efficient than specifying everything at the lowest level.

Example 5.2 Example using vector algebra function calls to perform the same operations as Example 5.1.

```
brhs = oldvi + vadto2 * (gstar .cross. oldvi) &
     + (vadt*raxis*local_jac) * dmid          &
     * add_stncl_vec(vilatest, THETA_DIR, 1,1)
```

– *Inflexibility* — the abstraction has to be able to provide the user with all the features and capabilities they need otherwise they may be forced to breakthrough the abstraction or abandon it completely. This creates "leaks" in the abstraction which destroy the portability advantages.

– *Ambiguity* — the abstraction has to be sufficiently well defined that operations produce the result intended consistently across implementations, and that there is no ambiguity about how to interpret the instructions.

The most flexible way to adapt CENTORI would be to rewrite the algorithms without making any reference to memory layout. By hiding information about the memory layout from the high level design it can be changed underneath without affecting anything above. This would require the creation of a library of functions and data types that CENTORI calls with a different implementation for each of the memory layouts. It allows each part to be developed separately without affecting the other. This is a replication of effort, but one considerably smaller and self-contained than maintaining different versions of the whole of CENTORI.

5.1.1 Mathematical Notation

The most obvious way to adapt CENTORI is to use a library expressed in mathematical notation, as used to describe CENTORI's fundamental equations in Section 1.4.1. Replacing verbose and complicated loops like those in Example 5.1 with functions and subroutines which look like mathematical operations allows the source code to be rewritten as in Example 5.2.

This example attempts to follow the mathematical notation as closely as possible using operator overloading. This is a Fortran language feature that allows user defined subroutines or function calls to be used in place of common operators like `+`,`/`,`-`,`*` or custom operators like `.cross.` on derived types.

The notation defines the relationship between fields like `oldvi`, `gstar` and `vilatest` without making reference to the layout of data in memory. Rewriting CENTORI to be memory

layout agnostic prevents CENTORI from becoming linked to any particular memory layout or implementation that it could not be changed.

5.1.2 Comparisons with other mathematical languages

Developing applications in a language derived from mathematics is not a new concept. There are many existing packages which provide numerical computing environments that CENTORI could have been described in. Matlab[78] is a popular example that is used in many fields to design and test algorithms written in mathematical notation. It has a highly expressive syntax and extensive library of functions and subroutines which can support a large variety of end users. Though it is a very useful prototyping tool, it is commonly used in environments where performance is not critical. It was not originally used to write CENTORI and continues not to be a suitable tool for the following reasons:

- *Performance* — Matlab is usually an interpreted language, though compilers do exist, which means that statements are converted to machine code on-the-fly. Interpreted languages like Matlab consistently perform poorly compared to compiled languages like Fortran.
- *Generality* — Matlab is a tool for solving very general mathematical problems, so has a wide scope and user base. Creating a domain specific library in Fortran, which may only be used for CENTORI, would allow it to be highly customised and optimised towards CENTORI's exact requirements.
- *Parallelism* — Fortran can use the message passing libraries like MPI, as well as shared memory techniques like OpenMP, to adapt code to run on multiple processors. Matlab's support for parallel processing of data is not as established as Fortran and MPI.
- *Proprietary* — Fortran is a widely supported standard with compilers from many different vendors. Though Matlab is a de facto standard for numerical computing environments, it is a proprietary application, so execution is limited to platforms this single vendor supports and the existence of a licence or licences to run on the system.
- *Legacy* — One of the most important reasons is that CENTORI already exists as a Fortran code, as such a large amount of effort has already been spent writing and testing the code in this form abandoning all the existing code and beginning to port the code from scratch would be grossly inefficient.

Example 5.3 Example subroutine showing repeated calls to the dot product function to calculate the dot product of two arrays of vectors.

```
function dotprod(a,b)
  real(kind(1.0D0)) :: dotprod
  dotprod = a%x*b%x + a%y*b%y + a%z*b%z
end function dotprod

subroutine multicall(c,a,b,n)
  real(kind(1.0D0)) :: c(n)
  type(vector) :: a(n), b(n)

  do i=1,n
   c(i) = dotprod(a(i),b(i))
  end do
end subroutine
```

5.1.3 Choosing the level

Most vector operations in CENTORI can be defined by a function that operates on individual vectors, either as binary or unary operands. When a function is written to perform the same action across an entire field of data a standard template is to make repeated calls to the original function for each member of the field. Reusing this code means there is only one definition of each operation meaning modifications only have to be performed in one place, however this incurs a performance penalty.

Consider the simple example of calculating the dot product of a generic array of vectors. Each array is a 1D version of the derived vector data type which has three double precision real components, x, y and z. As the dot product operation is defined on pairs of vectors it is easily implemented as a single function call; the dot product operation. The subroutine which performs the dot product operation on two vector fields calls the function for each pair of elements in the vector fields. Example 5.3 shows an implementation of this method.

Example 5.4 shows the same operation implemented for vector data in arrays only. It has exactly the same functionality but explicitly performs the dot product calculation in a line of the subroutine rather than making a function call. This would normally be discouraged as it means there are now two places the dot product is defined, increasing the overhead of maintenance but the reformulation has a significant performance advantage over the original, Example 5.3.

Full function calls introduce an overhead because the processor must add an entry on the "call stack" so it can return to the same state when the function exits. This is a set amount per function call so shorter functions that are called repeatedly have a much greater overhead,

Example 5.4 Example subroutine showing the dot product of two vector arrays being performed inline.

```
subroutine singlecall(c,a,b,n)
  real(kind(1.0D0)) :: c(n)
  type(vector) :: a(n), b(n)


  do i=1,n
    c(i) = a(i)%x*b(i)%x + a(i)%y*b(i)%y + a(i)%z*b(i)%z
  end do
end subroutine
```

proportionately, than longer functions that are called less frequently. The interruption of creating an element on the call stack impedes the flow of instructions and prevents vectorisation, creating stalls in the processor's pipeline to the detriment of overall performance, as explained in Chapter 3.

Inlining of functions avoids the function call and can be done automatically by compilers, theoretically combining the benefits of both techniques, however its success can depend upon the complexity of the function included and the quality of the compiler. Programs written in object-oriented languages like C++ and Java tend to use large numbers of small methods and include support in the language for inlining small functions.

Figure 5.1 confirms the performance advantage of the manual and automatic inlining over repeatedly calling a function on HECToR. On highest memory levels (i.e. smallest array sizes) inlining outperforms the repeated functions calls by up to 50%, dropping to approximately 20% on lower memory levels. There is also a small, but consistent, performance advantage in using the manually inlined method over the compiler method when on the lower memory hierarchy levels.

It appears that when performing the same operation on whole fields of vectors, as is the case in the majority of CENTORI, it would be best to use subroutines with the vector operations performed inline rather than repeatedly calling a function or subroutine. This could be achieved by inlining operations manually or automatically by the compiler, depending upon the performance advantage. By specifying relationships between whole vector fields rather than the relationship between individual points the interface could allow a greater variety of memory layouts to be implemented and provides the option to manually inline the operations if there is a performance advantage or limited compiler support.

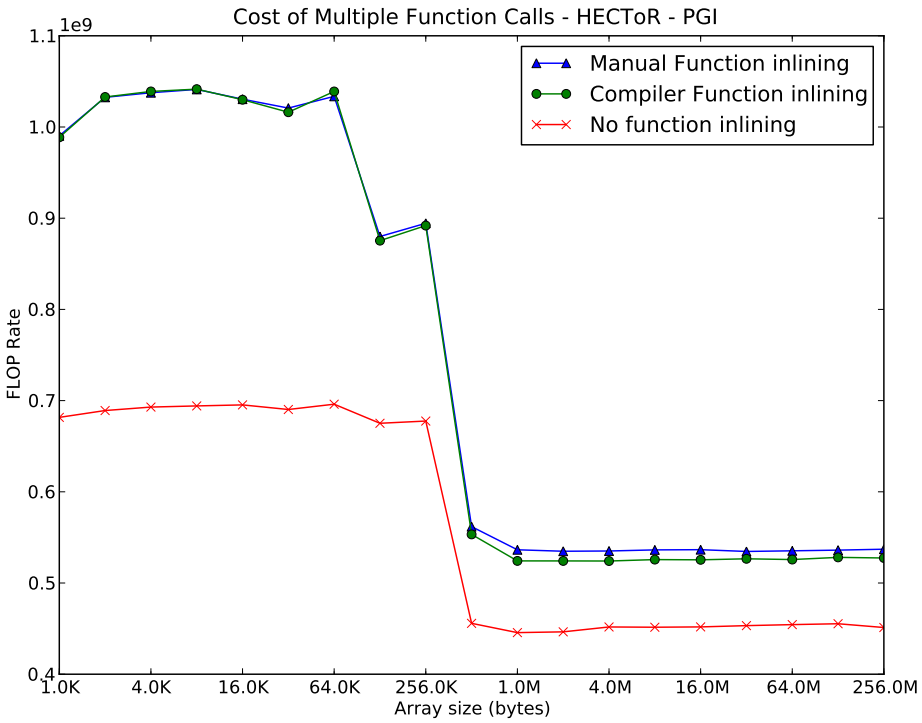


Figure 5.1: The performance (in Floating point operations per second) of each subroutine on HECToR for input arrays of different sizes.

5.2 Extending mathematical notation for CENTORI

Though mathematical notation is well defined and well understood by CENTORI's users and developers it cannot be included "as is" into CENTORI. Further definitions and restrictions are required to allow mathematical concepts to be incorporated into CENTORI's algorithms. Technical issues, peculiar to operating on a computer, also require addressing and limit functionality. These concerns and the impact they have on the interface are defined below.

5.2.1 Data types and mathematical operations

To hide the memory layout used by a particular implementation of the library, new data types are required for each of the fundamental objects used by CENTORI to encapsulate the data. Analysis of the original source code shows that four data types are required:

- *three dimensional vector fields* — for representing fields like velocity and the magnetic and electric fields. Originally, CENTORI represented this as an intrinsic 3D array of a derived type with nine double precision components and an integer tracker.
- *three dimensional scalar fields* — for fields like density and temperature. Originally represented as an intrinsic 3D array of native double precision floats.
- *two dimensional scalar fields* — quantities that only vary in the ψ and θ directions, e.g. the Jacobian. Currently represented as an intrinsic 2D array of native double precision floats.
- *one dimensional scalar fields* — quantities that only vary in the ψ direction e.g. flux surface averages. Current represented as an intrinsic 1D array of native double precision floats.

CENTORI does store single scalar values which are double precision floating-points values, however there is no advantage in encapsulating these items as they cannot be represented in any other form. The data types are only defined by a name, allowing them to be constructed in any way the implementation sees fit.

The interface and implementing library also need to provide functions that perform the following operations:

- (generally) addition, subtraction, multiplication and division (of non-vectors) for each data type and equivalents between each of the data types.

- (for scalar fields) square root, modulus, maximum value of individual fields.
- (for vector fields) vector product, scalar product, L2 norm.
- (derivative operations on appropriate fields) curl, divergence, gradient, partial derivatives of scalar fields.

Only the names, arguments and return types of each function are defined as well as a description of the operation, not the implementation. This allows them to be implemented in any way that conforms to the specification and will ultimately allow "portable" optimisation as the implementation changes underneath the specification. An exact specification of the data types and function calls required of an implementation is provided in Appendix A.

5.2.2 A Domain Specific Interface

The new library and interface need to be as well defined and precise as any programming language, but has the advantage of only being applicable to CENTORI so it does not require the same scope. By choosing an abstraction which is relatively high level, it has to express complicated concepts and operations succinctly and with sufficient scope that any future adjustments or extensions to CENTORI can be incorporated without undue effort.

This interface defines a very limited number of intrinsic data types, and a restricted set of predefined operations that manipulate them. It is only designed to perform the sequence of calculations that are relevant to CENTORI. As the library is an extension built on Fortran, which is already well defined, a lot of the operative details are the same (e.g. the operator precedence, assignment features) and additional language features are made available to simplify expressions. Operator overloading allows developers to nominate functions to replace standard mathematical operators acting on derived data types. This allows the `+` operator to be used on two vector field data structures, `a` and `b`, so a function call like this

```
c = add(a,b)
```

can be written as

```
c = a+b
```

The advantages of this approach are clearer when operating on multiple expressions, for example:

```
d = add(scale(a,b),c)
```

can be rewritten as

```
d = (a*b)+c
```

which corresponds more closely to typical mathematical equations. Additional operators can be defined for operations not part of Fortran's intrinsics, so operators like

```
c = a .cross. b
```

replace

```
c = cross(a,b)
```

which in turn improve the readability of expressions like

```
d = ((a+b)*c).cross.((d-e)*g )
```

instead of

```
d = cross(scale(add(a,b),c),scale(sub(d,e),g))
```

5.2.3 Data Parallelism

CENTORI is a parallel application with data spread over all the processors involved in the calculation. The mathematical expressions are oblivious to this and do not take into account this data parallelism; however the interface and library must do so. The vast majority of operations used do not require any communication between processors, however finite difference operations like curl or divergence require nearest-neighbour data and require the halos to be refreshed after each call.

This requires the library to make calls to the parallel modules to have the halos and boundary conditions updated, where some parameters are developer defined. This means more arguments have to be passed into finite difference operators than pure mathematical function requires:

```
a = curl(b)
```

becomes

```
a = curl(b, bc_axis_token , bc_edge_token)
```

Other than this high level change, the parallel nature of the application does not demand any further changes to the mathematical notation.

5.2.4 Breaking the abstraction

Structuring CENTORI's data types as black boxes without specifying anything about the data inside them leaves any library free to structure the data as it wishes. Apart from the difficulty in initialising a data object, it is almost impossible to predict all the ways the library may need to be used in CENTORI. Where it is not worth extending the library to include a particular operation, usually because it is small and occurs infrequently, there needs to be a standard way of accessing the data inside the data types without breaking the abstraction.

To achieve this a series of input and output routines are defined for each type, with the ability to read and write the internal data into or out of a standard buffer, either for a single point or an entire field. This allows individual points to be read and written to through function calls which, though inefficient, should not contribute significantly to the overall run time if done in small numbers. These extensions also include methods to extract fields of halo data from the data type and to put updated halo data back in, which is required in the parallel code.

Making these concessions in the abstraction allow the focus of the interface to be kept on core, computationally intense, time-consuming kernels whilst allowing the developer sufficient flexibility albeit for a small loss in performance. In cases where such operations are heavily used the specification of the interface has failed and probably should be extended to include the necessary functionality.

5.3 Memory Management

Expressions written in mathematical notation focus on conveying concepts clearly. They create, discard or rename objects without any cost leaving it to the interpreter (i.e. the human reading it) to take care of the accounting. Computers have limited memories and need explicit instructions to create new objects and explicit instructions to destroy them when they are no longer required. The interface and any implementation must be able to manage data objects in memory automatically without "leaking" memory. This section explains some of the requirements the interface places on implementation in the name of memory management and the reasons behind them.

5.3.1 Organising data in memory

Declared variables in a program are effectively labels given to a value held in the computer's memory. The location in memory of a variable cannot be changed by the user although its value can be. Pointers are a different kind of variable that can be referenced like any other variable, but can change the address in memory it is associated with. A pointer actually holds the address of the data it is pointing to which be changed by the program, allowing memory to be indirectly referenced.

This has numerous efficiency advantages. One of the most common operations in any programming language is to copy some values in memory to another location using the assignment operator, `'='`. For individual data types there is very little overhead in doing this, however for larger volumes, like arrays of data, it is much easier to use a reference to the position of data in memory using a pointer. If data only needs to be renamed, rather than copied in its entirety, it is quicker and more efficient to use a pointer.

5.3.2 Assignment

As the new data types place no restrictions on the way that data is held in memory, it is possible (and probable) that it will contain a pointer to the data in memory. This can lead to complex and unusual behaviour during assignments which is best avoided. Rather than copying all of the data referenced by a pointer, only the address will be copied, meaning two variables will now point to the same values in memory. This means that changes to data in the original pointer are making changes to data pointed to by the copy and vice-versa which is usually not required behaviour. For example

command	a (0x2710)	b (0x1830)	result
b=3.1	0.0	3.1	
a=b	3.1	3.1	
b=9.0	3.1	9.0	
print a	3.1	9.0	'3.1'

If `a` and `b` were ordinary data types, `a` and `b` would be the source code names for separate locations in memory, when the value of `b` is assigned to `a` it takes the value of `b` at that instant. When `b` is assigned a new value it has no effect on `a`, meaning the code will output the expected value 3.1 as shown above.

command	a (0x2710)	b (0x1830)	0x1240	0x8280	result
a=1.1	=>0x1240	=>0x8280	1.1	?	
b=3.1	=>0x1240	=>0x8280	1.1	3.1	
a=b	=>0x8280	=>0x8280	1.1	3.1	
b=9.0	=>0x8280	=>0x8280	1.1	9.0	
print a	=>0x8280	=>0x8280	1.1	9.0	'9.0'

If `a` and `b` are both pointers, the assignment will set `a` and `b` to point at the same address in memory, when `b` changes the value of this location it will change for `a` too. This leads to the potentially counter-intuitive result that the code will output what was the value assigned to `b`, even though it is supposed to be printing the value of `a`.

To avoid this kind of confusion the interface demands that the interface data types behave in the same way as native types in Fortran. Any implementation must make full, deep copies when making an assignment from a data type which already has a name. This prevents subtle and complicated bugs forming if the implementation uses pointers to store data.

When to apply copy by value

Enforcing the copy by value at every assignment is not necessary. There are some circumstances where the right hand side of an assignment does not already have a name, for example:

command	a (0x2710)	b (0x1830)	0x1230	0x8280	0x9310
b=3.1	NULL	=>0x8280	?	3.1	?
a=b	=>0x1230	=>0x8280	3.1	3.1	?
b=12.0*14.0	=>0x1230	=>0x9130	1.1	3.1	168.0

In the assignment on the second line, the right hand side is already assigned to the object `b`, and so the values pointed to by `b` have to be copied into a separate new location in memory for `a` to point to. The second assignment on line 3 is the result of a function (the `+` operator), so the output has no name and cannot be accessed through any other variable. In this case, `b` is allowed to directly point to the memory location with the result of the multiply function without copying the values.

As the majority of operations in CENTORI use assignments to store the results of expressions it is more efficient to perform the pointer assignment in every case where deep copies are not required, e.g. except when a copy assignment is made `(a = b)`. These rules are generally applied to make the interface's functions and data types behave in the same way as intrinsic data

types which programmers are very familiar with. By guaranteeing certain standards or methods of behaviour by the library implementation CENTORI can be rewritten with confidence that any implementation will produce the same overall results.

5.3.3 Repeated operands

Fortran requires that every argument to a function is unique, because operators on data types are converted function calls it means that the same operand cannot be used more than once in any operation, since doing so is undefined behaviour. If this behaviour is required objects should be manually assigned to a temporary value (which will perform a deep copy of the object) and the temporary and original object used instead e.g.

```
b = a + a ! Not allowed
! Instead, copy values to a tmp
tmp = a
b = a + tmp
```

Alternatively in this particular circumstance:

```
b = 2.0 * a
```

can be used instead.

5.3.4 Automatic type promotion

CENTORI requires an interface to four different types of data objects described in Section 5.2.1; vector fields, scalar fields, scalar planes and scalar profiles. Subroutines are defined to promote data types from a "smaller" type to a "larger" one, i.e. from a scalar field to a vector field. However, there are potentially multiple ways of "upgrading" a data type, of which CENTORI only requires two. Either by repeating the contents of the smaller data type across all the new dimensions which is the default position, or by placing the data in one particular section and filling the remainder with zeros.

Conversely smaller data types can be extracted from larger data types to provide only a subset of the information, for example it is possible to generate a scalar field from a single component of a vector field. These calls have to explicitly specify which section of the larger data type they require as there is no acceptable default solution.

Each of the datatypes needs function calls to allow them to interact, not only with objects of the same type, but also objects of different types. In many cases it may be useful to multiply a vector field by a scalar field, for example to find the momentum of the plasma. Rather than forcing the user to declare the types of operands (which can be complex and error prone if statements are long and compounded) the Fortran compiler can pick the correct function call which will then promote the smaller type to the larger type using the default copy method. If another arrangement of the data is required, the user will have to promote the data type manually.

5.3.5 Tidying up Temporaries

Modern computers are organised so there are two places where data can exist, the stack and the heap. The stack is used to store temporary values that are small and can be destroyed as soon as their useful life has passed, while the heap stores persistent values that have a longer life time (i.e. longer than an individual subroutine call) and have to be explicitly created and destroyed. If the location of a piece of memory in the heap is "forgotten" then that space in memory is lost forever and has "leaked" out of the program. Consistently leaking memory will eventually lead to the computer running out of memory entirely, forcing the program to exit prematurely.

In standard Fortran `+` is a binary operand, in one incarnation it takes two integer values as arguments and returns an integer as output. Every time the operation is performed a new object is created on the stack to hold the result (because addition is a "pure" function, and does not change the input arguments). It is very common to combine operations into compound structures where only the end result is required, for example the following:

```
a = (b + c) + d
```

This is actually two additions, with the first evaluating `(b+c)`, creating a new temporary value on the stack, which is then used as the argument for the second addition function, the result of which is stored as `a`.

```
tmp = b + c
a = tmp + d
```

This temporary value, `tmp`, is implicit and because it is on the stack the compiler is capable of recognising its fleeting lifespan, and discards it as it passes out of scope. The number

of temporaries can be quite large as statements are compounded, for example the following statement generates three intermediate temporary values.

```
c = sqrt((a*a)+(b*b))
```

The compiler is capable of managing these objects dynamically when they are located on the stack, however if the data is large or needs to be held for a longer period than a single operation it must be requested from the heap. All storage from the heap is uniquely allocated and cannot be recycled until it is released by the program. If the data types in CENTORI are going to be entirely generic the interface should assume that the data may be stored on the heap and referenced using pointers. This means additional functions are required for each data type to initialise and destroy data types.

Consider a very simple example:

```
b = initialise_sca_fld(1.0D0)
c = initialise_sca_fld(2.0D0)
d = initialise_sca_fld(3.0D0)


a = (b + c) + d
```

The overloaded operator + is being used to perform the operations, which is replaced internally with a call to the `add_scas` function. This makes the last statement equivalent to

```
a = add_scas(add_scas(b,c), d)
```

However there is an implicit temporary variable here, as + is a binary operator. To add all three terms a temporary value must be introduced.

```
tmp = add_scas(b,c)
a = add_scas(tmp,d)
```

Similarly to the scalar example, the temporary value is implicit in the original example, it is never explicitly named by the programmer and so cannot be accessed when it goes out of scope. However, it has been created by the `add_scas` function, potentially on the heap, as a store for the result of the function and if the data type contains a pointer to heap memory, when it goes out of scope the heap memory becomes unreachable causing a memory leak. Object Oriented languages like C++ and Java provide a destructor method for the data type which would be invoked when the `tmp` variable is destroyed, this could be used to deallocated the

memory, but unfortunately Fortran does not include this feature. Leaving such a memory leak in CENTORI is unacceptable, so a way of dynamically managing memory is a required part of the implementation.

There are many techniques for managing memory automatically, it is part of the design of many modern programming languages like Java and C#, and removes a significant overhead for programmer development. Fortran does not have automatic memory management for allocated memory, but does provide all of the tools needed for it to be implemented as part of the interface.

5.3.6 Reference Counting

Reference counting is one method of dynamically managing memory which can be simply implemented in Fortran. By keeping count of the number of pointers that refer to a memory location it is possible to detect when the memory has become unreachable and safely deallocate it. Reference counting is explained in detail by Aho et al.[79] where they explain the five points where referencing counting should intercept normal program flow:

1. *Object creation* — where reference counters are initialised to 1. Objects are created by any function which returns a result, e.g. addition, subtraction and scaling as well as the explicit initialisation functions.

2. *Parameter passing* — each object passed into a subroutine has its reference counter incremented by 1.

3. *Procedural returns* — when a subroutine returns all variables fall out of scope and cease to exist, each object must have its reference count decreased by 1. This requires the introduction of an unassign call that enforces this behaviour where the compiler cannot.

4. *Assignments* — the object on the left hand side of an assignment has its reference count decremented by 1 and the right hand side is increased by 1. Fortran allows the assignment operator to be overridden and replaced with a function which allows this functionality.

5. *Transitive Loss of Reachability* — when an object's reference count drops to 0 the object is destroyed and any child objects referred to have their references decremented by 1.

Every time references are decremented they are checked to see if the number of references has reached zero, and if this is the case the object is destroyed. To include this functionality in the original model additional types are created which have counters associated with data objects and are manipulated.

```
type smart_sca_fld
  real(kind(1.0D0)), dimension(:,:,:), pointer :: values => null
  integer :: refs = 0
end type

type sca_fld
  type(smart_sca_fld), pointer :: data
end type sca_fld
```

There are potential problems with reference counting that are discussed by Aho et al. These include cyclical references when groups of objects form a ring which can never be deallocated, as the number of references never goes to zero but they are unconnected to the main body of the program. Such situations are difficult to engineer in the case of CENTORI so have been disregarded.

Reference counting introduces an overhead by requiring extra computation at the start and exit of all routines and during every assignment operation. This can be exacerbated if small subroutines are used to implement the functionality, e.g. to increment and decrement counts and check for zero references and perform assignments. In most cases the computational complexity is much less than the cost of a function call as inlining can reduce the cost to almost zero.

To allow reference counting, or other automatic memory management techniques, to be included in any implementation the interface forces the user to explicitly unassign values before they fall out of scope, i.e. at the end of a subroutine. This allows the reference counting system to update the number of references and take appropriate action if they fall to zero.

5.4 Interface Definitions

Tables 5.1 and 5.2 contain the header definitions of the function and subroutines calls that the interface requires along with the basic data types that need to be implemented. This includes the operator interfaces that need to be defined to allow mathematical operations to be used on these native data types (described in brackets next to the subroutine description). A full description of the exact requirements of each function or subroutine call and how to use the library is contained in Appendix A.

Fundamental Data types

```
vec_fld - ! 3D collection of vectors in psi, theta and zeta
sca_fld - ! 3D collection of scalars in psi, theta and zeta
sca_pln - ! 2D collection of scalars in psi, theta
sca_prf - ! 1D collection of scalars in psi
```

Full Type I/O Functions

```
get_sca_fld(fld,buf)
get_scalar_plane(plane,buf)
get_scalar_profile(profile,buf)
get_vec_fld(fld,buf,coords)
set_scalar_plane(plane,buf)
set_scalar_profile(profile,buf)
set_sca_fld(fld,buf)
set_vec_fld(fld,buf,coords)
```

Halo I/O Operations

```
get_scalar_psi_halo_dn(fld,buf,xhalo)
get_scalar_psi_halo_up(fld,buf,xhalo)
get_scalar_theta_halo_dn(fld,buf,yhalo)
get_scalar_theta_halo_up(fld,buf,yhalo)
get_scalar_zeta_halo_dn(fld,buf,zhalo)
get_scalar_zeta_halo_up(fld,buf,zhalo)
get_vector_psi_halo_dn(fld,buf,xhalo,crd)
get_vector_psi_halo_up(fld,buf,xhalo,crd)
get_vector_theta_halo_dn(fld,buf,yhalo,crd)
get_vector_theta_halo_up(fld,buf,yhalo,crd)
get_vector_zeta_halo_dn(fld,buf,zhalo,crd)
get_vector_zeta_halo_up(fld,buf,zhalo,crd)
set_scalar_psi_halo_dn(fld,buf,xhalo)
set_scalar_psi_halo_up(fld,buf,xhalo)
set_scalar_theta_halo_dn(fld,buf,yhalo)
set_scalar_theta_halo_up(fld,buf,yhalo)
set_scalar_zeta_halo_dn(fld,buf,zhalo)
set_scalar_zeta_halo_up(fld,buf,zhalo)
set_vector_psi_halo_dn(fld,buf,xhalo,crd)
set_vector_psi_halo_up(fld,buf,xhalo,crd)
set_vector_theta_halo_dn(fld,buf,yhalo,crd)
set_vector_theta_halo_up(fld,buf,yhalo,crd)
set_vector_zeta_halo_dn(fld,buf,zhalo,crd)
set_vector_zeta_halo_up(fld,buf,zhalo,crd)
```

Assign Functions

```
assign_vector(lhs,rhs)
assign_scalar(lhs,rhs)
assign_plane(lhs,rhs)
assign_profile(lhs,rhs)
```

Point wise I/O Functions

```
get_point_scalar(fld,pos)
get_point_plane(fld,pos)
get_point_profile(fld,pos)
get_point_vector(fld,pos,buf,coords)
set_point_scalar(fld,pos,buf)
set_point_plane(fld,pos,buf)
set_point_profile(fld,pos,buf)
set_point_vector(fld,pos,buf,coords)
```

Partial Type I/O Functions

```
get_plane(fld,zeta)
get_psi_profile(plane,theta)
get_component(vec,crds,cmp)
set_component(vec,sca,crds,cmp)
set_psi_profile(pln,prf,theta)
set_plane(fld,pln,zeta)
```

Type Conversion Functions

```
sca_to_vec(a,coords,cmp)
pln_to_sca(a,zeta)
prf_to_pln(a,theta)
prf_to_sca(a,theta,zeta)
prf_to_vec(a,coords,cmp,theta,zeta)
pln_to_vec(a,coords,cmp,zeta)
```

Initialisation Functions

```
initialise_vec_fld_cnst(crds,cnst)
initialise_vec_fld_buff(crds,buf)
initialise_sca_fld_cnst(constant)
initialise_sca_fld_buff(buf)
initialise_pln(cnst)
initialise_prf(cnst)
```

Coordinate Functions

```
get_coords(in)
to_contra(in)
to_coords(fld,coords)
to_covariant(in)
to_physical(in)
```

Table 5.1: The fundamental data types defined by the interface and the system level functions to operate on them. `vec` refers to operations on 3D fields of vector data, `sca` to operations on 3D fields of scalar data, `pln` to operations on 2D fields of scalar data and `prf` to operations on 1D fields of scalar data.

Addition Functions

```
add_plns(a,b)                    (+)
add_prfs(a,b)                    (+)
add_scas(a,b)                    (+)
add_vecs(a,b)                    (+)
add_vec_sca(a,b)                 (+)
add_vec_pln(a,b)                 (+)
add_vec_prf(a,b)                 (+)
add_sca_vec(a,b)                 (+)
add_sca_pln(a,b)                 (+)
add_sca_prf(a,b)                 (+)
add_pln_vec(a,b)                 (+)
add_pln_sca(a,b)                 (+)
add_pln_prf(a,b)                 (+)
add_prf_vec(a,b)                 (+)
add_prf_sca(a,b)                 (+)
add_prf_pln(a,b)                 (+)
```

Add Constant Functions

```
add_cnst_sca(cnst,sca)           (+)
add_cnst_prf(cnst,prf)           (+)
add_cnst_pln(cnst,pln)           (+)
add_sca_cnst(sca,cnst)           (+)
add_prf_cnst(prf,cnst)           (+)
add_pln_cnst(pln,cnst)           (+)
```

Mean Square Functions

```
mean_square(ftot,ffl,fsq,fflsq,frsq)
mean_squarev(ftot,ffl,fsq,fflsq,frsq)
```

Multiply Functions

```
mult_vec_sca(a,b)                (*)
mult_vec_pln(a,b)                (*)
mult_vec_prf(a,b)                (*)
mult_sca_vec(a,b)                (*)
mult_sca_pln(a,b)                (*)
mult_sca_prf(a,b)                (*)
mult_pln_vec(a,b)                (*)
mult_pln_sca(a,b)                (*)
mult_pln_prf(a,b)                (*)
mult_prf_vec(a,b)                (*)
mult_prf_sca(a,b)                (*)
mult_prf_pln(a,b)                (*)
```

Multiplication Functions

```
mult_scas(a,b)                   (*)
mult_plns(a,b)                   (*)
mult_prfs(a,b)                   (*)
mult_vecs(a,b)                   (*)
```

Integration Functions

```
flux_surface_integral(sca,axis,edge)
flux_surface_average(sca,axis,edge)
toroidal_average(sca,axis,edge)
volume_integral(sca,axis,edge)
volume_integrate_profile(prf,axis,edge)
integrate_profile(prf,ax2ed,init)
integrate_profile_global(prf,ax2ed,init)
integrate_in_zeta(sca)
```

Stencil Add/Subtract Functions

```
add_stncl_sca(in,dir,up,down)
add_stncl_pln(in,dir,up,down)
add_stncl_prf(in,dir,up,down)
add_stncl_vec(in,dir,up,down)
sub_stncl_sca(in,dir,up,down)
sub_stncl_vec(in,dir,up,down)
sub_stncl_pln(in,dir,up,down)
sub_stncl_prf(in,dir,up,down)
```

Subtraction Functions

```
sub_plns(a,b)                    (-)
sub_prfs(a,b)                    (-)
sub_scas(a,b)                    (-)
sub_vecs(a,b)                    (-)
sub_vec_sca(a,b)                 (-)
sub_vec_pln(a,b)                 (-)
sub_vec_prf(a,b)                 (-)
sub_sca_vec(a,b)                 (-)
sub_sca_pln(a,b)                 (-)
sub_sca_prf(a,b)                 (-)
sub_pln_vec(a,b)                 (-)
sub_pln_sca(a,b)                 (-)
sub_pln_prf(a,b)                 (-)
sub_prf_vec(a,b)                 (-)
sub_prf_sca(a,b)                 (-)
sub_prf_pln(a,b)                 (-)
```

Subtract Constant Functions

```
sub_sca_cnst(sca,cnst)           (-)
sub_prf_cnst(prf,cnst)           (-)
sub_pln_cnst(pln,cnst)           (-)
sub_cnst_pln(cnst,pln)           (-)
sub_cnst_sca(cnst,sca)           (-)
sub_cnst_prf(cnst,prf)           (-)
```

Square root functions

```
sqrt_scalar(fld)                 (sqrt)
sqrt_profile(fld)                (sqrt)
```

Scale Functions

```
scale_pln(pln,constant)          (*)
scale_prf(prf,constant)          (*)
scale_sca(sca,constant)          (*)
scale_vec(fld,constant)          (*)
scale_pln_opp(constant,plane)    (*)
scale_prf_opp(constant,profile)  (*)
scale_vec_opp(constant,vec)      (*)
scale_sca_opp(constant,sca)      (*)
```

Inversion Functions

```
invert_scalar(constant,in)       (/)
invert_plane(constant,in)        (/)
invert_profile(constant,in)      (/)
```

Partial Derivative Functions

```
d_psi(fld,dpsi)
d_psi_profile(profile,dpsi)
d_theta(fld,dtheta)
d_zeta(fld,dzeta)
```

Vector Algebra Functions

```
curl(a,bc_axis,bc_edge)        (.curl.)
div(a,bc_axis,bc_edge)          (.div.)
grad(a,bc_axis,bc_edge)       (.grad.)
dot(a,b)                        (.dot.)
cross(a,b)                    (.cross.)
l2(a)
max_sca_fld2(a,b)
abs_scalar(sca_fld)
```

Table 5.2: Mathematical functions defined by the interface to operate on the data types defined in Table 5.1 along with operator interfaces to be defined in brackets. `vec` refers to operations on 3D fields of vector data, `sca` to operations on 3D fields of scalar data, `pln` to operations on 2D fields of scalar data and `prf` to operations on 1D fields of scalar data.

5.5 Performance Implications of Using the Interface

By creating this interface CENTORI can be rewritten to use any implementation that complies with the definition. This allows different implementations to be created, specifically versions which are optimal on particular platforms, and details such as optimised memory layout and optimised operations can be included to accelerate CENTORI. The interface has been left sufficiently generic that there are a broad number of possible implementations, yet is specific enough that CENTORI should produce the same result whichever implementation is used. However there are potential performance implications.

The interface only defines functions that perform single operations on entire fields of data. In a simple implementation each library function is implemented as a function that performs the required operations on the input array arguments and writes the result to output arrays. This means all the results must be calculated and written to the output array in their entirety before any other operations can be performed. Consider the following example

```
a = (c + d) * k
```

This operation involves two functions, `add` and `scale`, and will be broken down by the compiler to an operation like this:

```
tmp = add(c,d)
a = scale(tmp, k)
```

In a 1D case, each operation would be implemented as a function that performs a loop over the input arrays and writes to the output. This requires a temporary array, `tmp`, to store the intermediate values.

```
! Add function
do i=1,n
  tmp(i) = c(i) + d(i)
end do

! Scale function
do i=1,n
  a(i) = tmp(i) * k
end do
```

However these loops can be fused into a single loop and the array temporary replaced by a single scalar temporary value. This increases the computational intensity by doubling the number of operations performed for each access to memory. This will perform better because the temporary value can be held in register, rather than being written back to the memory hierarchy which is likely to happen if `n` is large.

```
do i=1,n
  tmp = c(i) + d(i)
  a(i) = tmp * k
end do
```

Using two function calls is a poor use of cache, though as long as the operations exist as two different functions the loops can never be fused together. Creating even more compound kernels like a "scaleadd" function is not as flexible as having separate add and scale functions and eventually leads back to complicated and non-portable code as in Example 5.1.

Ideally there would be a way to implement the library without extending the interface that would make better use of the cache architecture of the processors.

Chapter 6

Optimising Performance of the Library

Chapter 5 describes an interface to a library that can be used to express the fundamental equations of CENTORI in a uniform way that masks the underlying implementation and data structures from the programmer. It does not describe any particular implementation though it identifies some shortcomings with temporary data that a naive implementation of the interface may encounter that would reduce the performance of the code on most architectures. This chapter gives an overview of how CENTORI was converted during this study to use the new library and describes two optimised implementations of the library that target HECToR.

6.1 A Reference Implementation

Converting the original CENTORI to use the interface required a substantial rewrite of the original code base. To make the task more manageable the integration was performed in stages which allowed incremental updates with frequent testing. Each line of the existing code was replicated as a new line of code in the same file using the new interface. After each directly comparable statement (or statements) the results of the new and old code were compared by calculating the average root mean square relative difference between the data fields (not including the halos). If this value was below a certain threshold ($< 10^{-13}$) the fields were deemed to be identical and the result was recorded in an output file. This method produced three important results:

– A new version of CENTORI which used both the old vector notation and the new vector notation to evolve the model, which tested each call of the library against the original

code. This is an extremely useful tool for debugging new implementations of the library and was very useful in testing and evaluating the first reference version of the library.

– A complete version of CENTORI that expressed the core equations using the new interface only (after the original code was removed from the joint version). This version can use any implementation of the library and became the operational standard after testing and evaluation. This version could only be compared with previous version by comparing the final results, but was found to have the same results as the original code.

– A first reference implementation of the library that performed all of the operations defined by the interface (as required by the current CENTORI) that had been tested and found to produce the same results as the original version of CENTORI.

The reference version had subroutines and functions for each method required by the interface that perform the required calculations using methods similar to the naive kernels in Chapter 4. It defined the appropriate data types as pointers to arrays of double precision real data of the appropriate dimension, arranged using the "array" memory layout format for the vector field data type. This code became the definitive reference implementation of the library rather than a high performance version and so includes no additional hand optimisations. It complements the interface by providing the exact definition of the functionality required for CENTORI.

6.1.1 An optimised version of the library

The reference implementation was not customised to perform optimally on a particular platform (in fact the resulting combined version of CENTORI did not degrade performance compared to the original code on HPCx, HECToR and HPC-FF). One of the major advantages of the library is that its implementing code can be altered and adapted to the target architecture without requiring any further changes to the main CENTORI and vice-versa. The next step in optimising CENTORI was to create a new library which replaced the naive code with the subroutines which performed best for the memory layout and architecture combination from the benchmarks in Chapter 4.

This first optimised implementation was designed for use on HECToR with the PGI compiler for $10 \times 10 \times 10$ local problem sizes; it used the "array" format as it was an adaptation of the initial reference implementation, for which many parts remain the same. The individual high performance functions were adapted from the fastest performing kernels with the appropriate

memory layout, platform and data sizes as measured in the benchmarks in Chapter 4. Due to the similarity of the kernels with the required operations the source code only had to be lightly modified. Operations which were not represented in the benchmarks were written and hand optimised using the experience and evidence gained from the benchmarking process and its results.

6.2 Intermediate Temporary Values

Implementing the functionality of the interface as a series of Fortran functions requires a lot of data to be stored in temporaries. These temporaries could be of any type, maybe full vector fields or scalar fields. Each of these temporary data structures occupies space in the caches, and if they grow too large the final values of the objects may evict the first values when the data is written back to main memory.

6.2.1 Eager Evaluation

Fortran is a procedural language, it compiles the source code into a sequence of instructions which perform a set sequence of tasks. It uses a standard "eager" evaluation scheme, meaning that if a function in the interface is called it must perform all the necessary calculations in one go. Every function that is called is executed in full before the function returns to the caller and continues processing the next instruction. Each function is called in isolation and the only interaction between functions is through the values passed in and out via the arguments. Functions that will be called later can have no influence on the current function or any that have been called previously.

When performing an operation on an entire field of scalar or vector data it must read all of the input arguments and produce all of the output data before returning to the caller. This forces programs which use functions to perform operations on large arrays (as the new CENTORI does) to stream large volumes of data. This can potentially evict the first elements of an array from the cache by the time last values are accessed (assuming incremental access patterns).

A $34 \times 34 \times 34$ vector field in CENTORI contains at least 921 KB of data, so calling a function to add two vector fields and create a new output vector field requires over 2.70 MB. This is larger than each core's share of the L3, the L2 and L1 caches combined on the quad

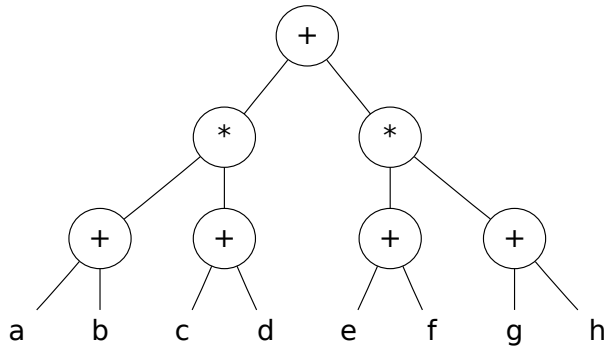


Figure 6.1: A tree of binary operands for the operation `((a+b)*(c+d))+((e+f)*(g+h))` showing 6 intermediate temporaries, 8 inputs and 1 output.

core Opterons in HECToR, so it is very likely that a large proportion of the results will have been evicted from the caches into the main memory. In most cases the result of a function will form part of the input to subsequent operations, but if the first elements of the result have been evicted from the cache into main memory they will need to be reloaded from the main memory with all the limitations of memory bandwidth and latency that this implies.

6.2.2 Growing Temporary Data Volumes

Nearly every major calculation in CENTORI is a combination of basic unary (single input) or binary (double input) operations. Any calculation that is more complicated than a single operation will require intermediate temporary arrays to hold the output which forms the input to the next operation. Figure 6.1 shows a simple symmetrical tree of the binary operations which requires 6 intermediate temporaries from 8 input operands during the calculation of a single output. As more operations are combined, the number of intermediaries increases with any symmetrical binary tree of n levels requiring $2^n - 2$ intermediates for 2^n inputs being reduced to a single output.

In reality CENTORI's Abstract Syntax Trees (AST) do not form perfect, symmetric binary trees, instead they mix binary and unary operations on both scalar and vector operations. Figure 6.2 is an example of the AST from CENTORI, which takes 7 vector fields and 4 scalar fields as input and produces a single vector output. Performing this calculation requires holding 17 vector and 10 scalar temporaries during the course of the calculation.

If these temporaries are very small, e.g. only one or two double precision values, they can

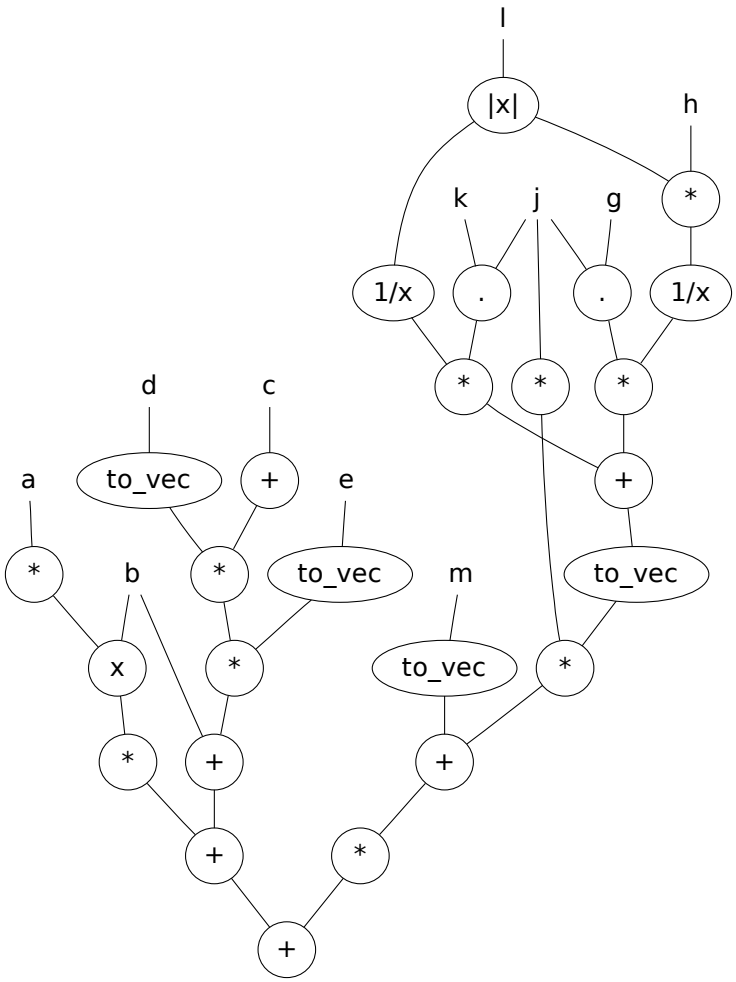


Figure 6.2: An example tree of operations extracted from CENTORI, formed from unary and binary operations and scalar and vector fields. There are 7 vector and 4 scalar inputs producing 1 vector output requiring 17 vector and 10 scalar temporaries.

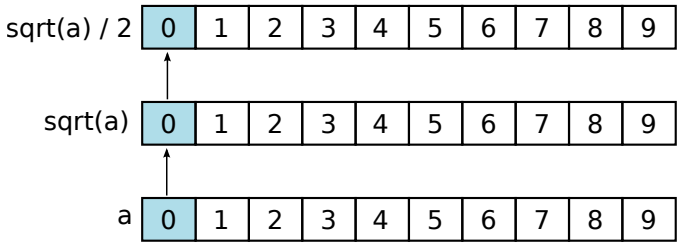


Figure 6.3: An example sequence of operations which has only local data dependencies, including a full size intermediate temporary value.

be stored in registers and the cost of manipulating them is practically zero. This can only be created by the compiler when there is a source code implementation of the algorithm that specifies the memory layout of the data structures, as in Example 5.1, precisely the situation the interface is designed to prevent. If the operations are performed by functions, as described by the interface, then the intermediate temporary values must be full fields which will be very large, e.g. for an array sized $34 \times 34 \times 34$ the data in Figure 6.2 represents 7.50 MB of input data, 0.90 MB of output data and 18.3 MB of temporary data, a total of 26.7 MB. This forces the processor to operate completely out of main memory, where it is slowest, rather than the much faster caches which are intended to hold transitory data. Figure 6.3 shows the dependency relationship between values in array temporaries for local data volumes.

6.3 Reducing Intermediary Temporary Data Volumes

Using highly optimised libraries to perform common operations is a well established technique, classical examples are the Basic Linear Algebra Subprograms used in linear algebra that was discussed in Section 4.3. Having to hold data in large intermediate arrays between calls to fast library functions has been acknowledged as a poor use of the memory hierarchy. Projects exist that have developed domain specific compilers to generate highly optimised functions for use in libraries that attempt to remove these limitations.

The “Built to Order BLAS”[80] project translates annotated Matlab statements into optimised C++ which can be called from a standard application. It aims to combine multiple calls to a library like BLAS together as a single subroutine which can take advantage of fusing loops and other optimisations that cannot be achieved with the standard call. The project achieves significant speed ups (100% to 130%) over the same operations implemented using separate native BLAS libraries on operations tested. This approach may be useful for producing the

more complicated kernels in CENTORI like curl or divergence, however does not offer help with the interface as defined, i.e. it would be most beneficial combining sequences of operations into a single optimised function call.

The SPIRAL project[81], attempts to achieve a similar goal for Digital Signal Processing (DSP) transformations, producing automatically optimised code translated from a domain specific language. This project has produced automatically optimised libraries for Fourier transforms and other trigonometric functions which have outperformed human tuned libraries. While the broad aims of this project are similar to CENTORI's interface and library project it has not been applied in this case.

Instead of trying to adapt a third party tool to optimise or remove the intermediate temporary values, it is possible to make best use of the memory hierarchy available using standard Fortran and a technique known as "stripmining"[82].

6.3.1 Stripmining

If the intermediate temporary values were stored in the caches they would be available for reuse much more quickly when they were required. Reducing temporary sizes can be achieved globally by storing a smaller number of data points locally (i.e. decompose the problem over a large number of processors) or by only storing a small part of each temporary at any one time.

Assuming all of the operations in CENTORI are local functions the AST can be evaluated for any grid point without depending on the results of neighbouring point. (if not it can be handled by loop skewing see Section 6.4). Breaking a large field into smaller strips or tiles of data means it is possible to perform all the operations in the AST on the data in the strip whilst requiring only a fraction of the temporary data, which may be small enough to fit into a higher cache level. This can be repeated for each strip in the larger field and the intermediate temporary values can be reused because the data they contain is transitory.

Overall, the calculation should complete more quickly due to the increased use of the caches instead of main memory. Using the example from Figure 6.2, if the calculation used strips sized $34 \times 34 \times 1$, the temporary data would only occupy 0.54 MB (compared to 18.3 MB), which is much more likely to fit on a processor's cache architecture. This technique is known as "strip mining" or "tiling" and is an established technique for improving the ratio of floating-point operations against the number of data loads from main memory by reducing the number of memory accesses between data reuse[83, 82, 84].

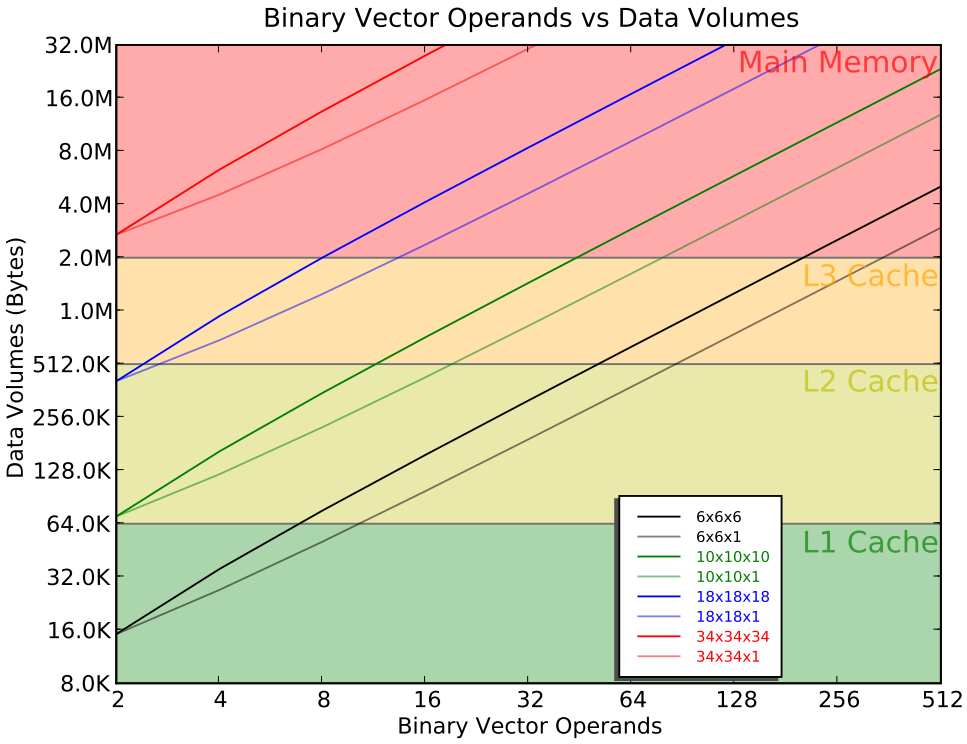


Figure 6.4: Graph showing the growth in data volumes as the number of binary operands increase on vector fields. Bold lines indicate data volumes with full temporaries, lighter lines the volume using smaller temporaries. Colour bandings represent the memory hierarchy of an Opteron CPU.

Example 6.1 Two loops using a temporary array to store the intermediary values.

```
do i=1,n
  tmp(i) = sqrt(a(i))
end do

do i=1,n
  out(i) = tmp(i) / 2.0
end do
```

Example 6.2 Folding the common outer loop into a single loop.

```
do i=1,n
  tmp(i) =sqrt(a(i))
  out(i) = tmp(i) / 2.0
end do
```

Figure 6.4 shows the total volume of data stored in the inputs, outputs and intermediates when evaluating a balanced binary tree of vectors when using full sized temporaries versus reusing smaller 2D temporaries. These are displayed for the typical array sizes used in CENTORI for a varying number of input operands. The sizes of the caches on a quad core Opteron (used in HECToR) are included for comparison. Reduced temporary sizes show significant reductions in the total amount of memory used by the calculation meaning memory performance moves from a lower memory hierarchy level to a higher one.

This example is modelled on the growth of symmetric binary trees, while CENTORI's real operations generate more temporary data objects because they are unbalanced trees and use unary operands (e.g. Figure 6.2). This results in even greater reductions in total memory use when using smaller temporaries.

6.3.2 Example stripmining

Stripmining can be implemented in Fortran by introducing an extra outer loop in most iterative calculations. The external loop goes through each of the individual strips to be processed. Example 6.1 shows two loops that perform two separate calculations, where the intermediate temporary value is stored as an array. There are `n` stores between the temporary value being calculated and the result being reused, so if `n` is large enough the `tmp(1)` may have been evicted from cache by the time `tmp(n)` has been calculated. In Example 6.2 the outer loop has been collapsed so the delay between a temporary being stored and reused is eliminated.

Unless the intermediary value is required in a future calculation there is no need to store

Example 6.3 The temporary array is replaced with a scalar value.

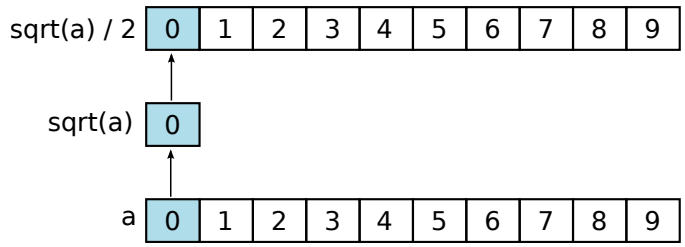
```
do i=1,n
  tmp=sqrt(a(i))
  out(i) = tmp / 2.0
end do
```

it after the next iteration of the loop, instead only a single result needs to be stored as in Example 6.3. This does not reduce the total amount of data read and written (there are still `n` read and `n` writes), it reduces the number of locations in memory accessed to one. Each time a new memory address is written to (even if it is not read) a write miss may be incurred if it is not already cached because the system must load an entire cache so that single value can be altered. Reusing the same location(s) in memory avoids this overhead and is generally the fastest option. Figure 6.5 illustrates how the temporary value is reduced in size.

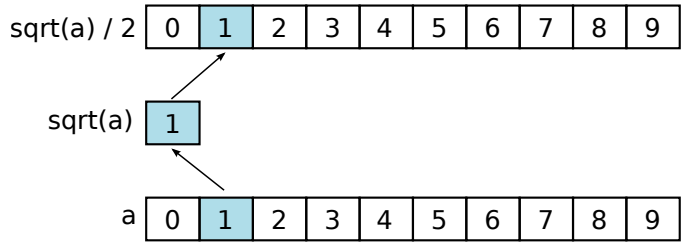
6.3.3 Sizing strips

Stripmining does not reduce the number of calculations performed, or the amount of data processed. It does, however, reduce the number of floating-point operations between a result being stored in memory and it being reused as input in the next calculation. Reducing this delay increases the probability that the data is still resident in the cache when required which decreases the access time for it. The scalar and vector temporaries in the stripmining scheme can be any size, from scalar values of size 1 to arrays the same size as full fields. As has already been noted, using implicit temporaries allows the compiler to hold temporaries as scalars in registers which produces the best performance but requires a heavy integration between the algorithm and the memory layout.

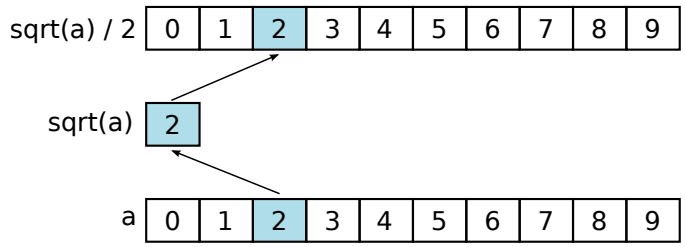
Strip size is a tunable parameter that affects the performance of the code in a variety of ways. Very small blocks are likely to perform poorly because even though the temporary data is always in a high level of cache there will be many individual blocks which incurs an overhead in the processing. Using a larger strip reduces the overhead and allows the processor to use vector instructions with fewer interruptions from function calls or loops, but risks evicting valuable data from the caches before it can be read. Tuning the size of blocks will be a significant factor in optimising CENTORI's performance on each architecture.



(a) The first iteration of the operation, the temporary will incur a write miss if the value is not in cache.



(b) Second iteration, the temporary value is overwritten (no write miss as temporary already in cache).



(c) Third iteration (no write miss).

Figure 6.5: Step by step diagram of performing a calculation reusing the temporary location.

Example 6.4 Two loops which cannot be immediately fused due to data dependencies.

```
do i=1,n
  tmp(i)=sqrt(a(i))
end do

do i=2,n-1
  out(i) = (tmp(i+1) - tmp(i-1)) * 0.5
end do
```

6.4 Dealing with Data Dependencies

Section 6.3.1 describes using stripmining to optimise operations that are only dependent on local data, e.g. the value of the result at coordinate `(1,3,5)`, `out(1,3,5),` is only dependent on the values of the input arrays at `a(1,3,5)` and `b(1,3,5)`. However many operations in CENTORI, like finite differences, are not locally dependent and require input data for nearest neighbour grid points, which introduces more complicated data dependencies into a stripmining scheme.

6.4.1 Data Dependencies and Loop Skewing

Operations that only have local data dependencies mean loops can be fused and operations performed in any order (or even parallelised). Operations that have wider data dependencies cannot have their loops easily fused as in Examples 6.1 and 6.2. Finite difference operations require data from the nearest neighbours of the coordinate being processed to calculate the correct result. Example 6.4 shows a 1D finite difference operator that cannot be directly fused with the original loop.

In this example the temporary has to hold all `n` values, as the second finite difference loop requires the results of the `sqrt` operation to be calculated before it can be executed. This reverts the problem back to using large intermediate temporaries which, as outlined in the previous sections, is undesirable. So adapting these operations to use smaller temporaries would be a big improvement for cache reuse of the code.

One method, shown in Example 6.5, is to calculate both the “up” and “down” values for the finite difference for each iteration of the outer loop. This doubles the amount of computation performed on most data points, once as the “up” point and once as the “down” point. Though this reduces the size of the temporary it introduces additional unnecessary repeated calculations which may be very costly when the input arguments are more complex than a single unary

Example 6.5 The dependent values are calculated in advance of the finite difference operation. This means the `sqrt` is called twice for a large number of central points.

```
do i=2,n
  tmp1=sqrt(a(i+1))
  tmp2=sqrt(a(i-1))
  out(i) = (tmp1 - tmp2) * 0.5
end do
```

Example 6.6 Non repeating dependency implementation using modulus address to minimise the temporary sizes.

```
! tmp is now an array of size 3
tmp(1) = sqrt(a(1)) ! initialise the first two values of
tmp(2) = sqrt(a(2)) ! tmp with the result of the calculation

do i=2,n-1
  tmp((mod(i+1,3))+1) = sqrt(a(i+1)) ! Update the leading tmp
     with the calculation
  out(i) = (tmp((mod(i+1,3))+1) - tmp((mod(i-1,3))+1)) * 0.5
end do
```

operator like `sqrt()`. It is therefore usually unfavourable to trade off floating-point operations for memory accesses when there is potentially such a large imbalance.

A more sophisticated method of dealing with the situation is described in Example 6.6 which only calculates each temporary value once whilst also storing them in a temporary array which is smaller than a full field. This method stores the temporary values in an array which is the size of a usual strip but includes extra strips at the beginning and end of the temporary to hold the values of the "up" and the "down" parts. The temporary values are referenced by taking the modulus of the required index and the width of the temporary object. Having these two extra values means that before entering the loop the first two values of the temporary have to be calculated in advance so any subsequent calculations are for a single strip, the $(i+1)^{\text{th}}$ value (the "up" point).

This technique can be chained when there are multiple operations requiring dependencies, with the temporary being increased in size as more halo points are required. This increase in the volume of temporary data needs sophisticated book keeping code to keep track of the dependencies of each object, especially when dealing with more than one operand. The code in Example 6.6 becomes dominated by the book keeping rather than the calculation, making it difficult to modify or adjust this code.

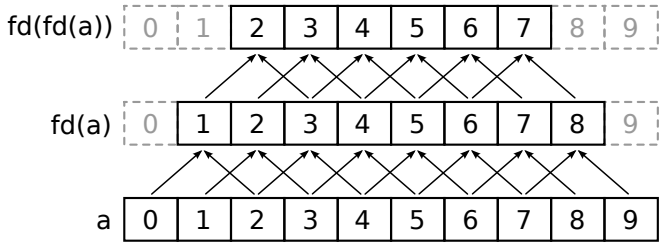
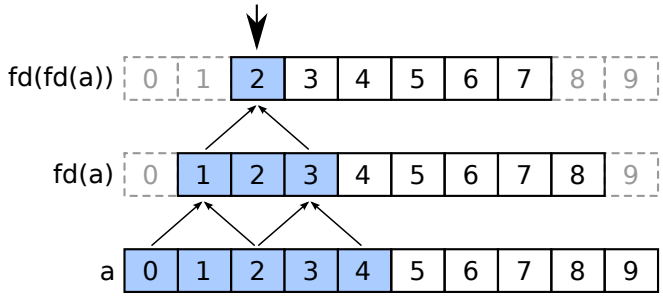
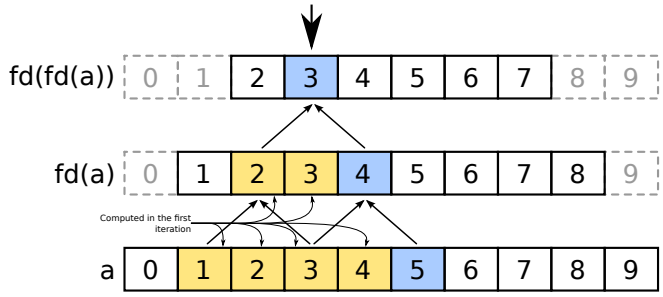


Figure 6.6: A dependency graph for a sequence of derivative operations.



(a) First iteration: All dependencies are evaluated for the top level strip being processed.



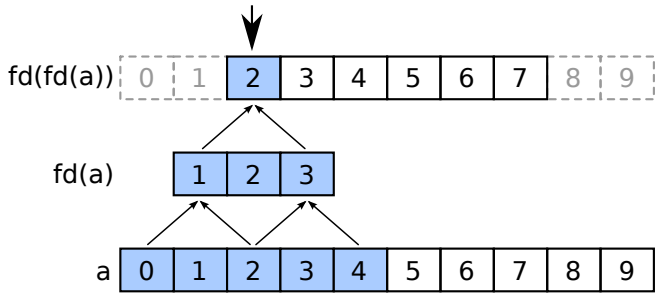
(b) Subsequent iterations: only require the additional dependencies to be evaluated so all subsequent calculations are the same size as strip.

Figure 6.7: Strip mining with dependencies.

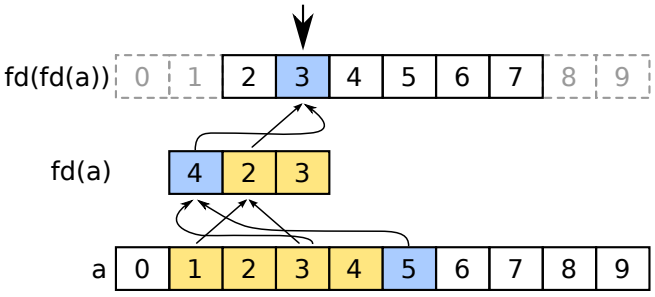
6.4.2 Managing Multiple Dependencies

Figure 6.6 shows the dependency relationship between points when performing a generic finite difference operation in one dimension, `fd()`. The result depends upon the values of two points from the input field which are separated by 3 points. If we adopt the approach outlined in the section above, the first output value requires all of the dependent points to be available in advance, including all points in the intermediate temporary. The highlighted values in Figure 6.7a are the ones that must be computed in advance; Figure 6.7b shows how subsequent values can be calculated without having to recompute the entire dependency tree.

Each global index is mapped to the smaller temporary index using the modulus operation of the address with the width of the temporary value. All of the dependent values of the



(a) First iteration: All dependencies are evaluated for the top level strip, the intermediary has the correct data in the standard order.



(b) Subsequent iterations: only additional dependencies evaluated, however data is wrapped around the intermediary array to overwrite unnecessary data.

Figure 6.8: Strip mining with dependencies and a minimal intermediary.

first result have to be computed in advance, see Figure 6.8a, and each subsequent iteration is skewed so it only has to calculate one new value for each temporary result in a "wavefront" style approach[85, 86]. Data is overwritten when it is no longer required and is replaced by the latest values needed to calculate the correct result as show in Figure 6.8b. This technique minimises the amount of data held in temporaries to the minimum amount required to perform the calculation which increases the probability of the local data being held in one of the caches when needed.

6.4.3 Reconciling cache blocking with the interface

The interface described in Chapter 5 was designed to separate the memory layout from the high level application logic to allow different implementations to be used with the same high level CENTORI, and for the high level algorithms in CENTORI to be changed without thought of the implementation. This was done by defining a series of data types and function calls to communicate between the layers.

The original functions which mimic standard vector algebra notation do not support this cache blocking technique which needs an external loop and would require several extra argu-

Example 6.7 Same code as Example 6.3 using subroutine calls to abstract the memory layout from the stripmining.

```
! If mod(n,m) == 0
do i=1,n,m
  call sqrt(tmp, a, i, m, n)
  call scale(out, temp, 2.0, i, m, n)
end do
```

Example 6.8 Same operation as Example 6.7 expressed as calls to the interface.

```
out = sqrt(a) / 2.0
```

ments to be passed into each function to keep track of which strip is being processed. When dealing with operations with data dependencies which require the nearest neighbours to be calculated it makes management of the size of temporaries even more complicated. Example 6.7 shows how the interface might be adapted to incorporate stripmining. However this is still much more complicated and less elegant than the original notation of the interface in Example 6.8. It also binds the high level applications into using a cache blocking library, even if it has no performance advantage, another violation of the aims of the original interface.

Ideally, CENTORI could keep the "nice" features of the interface - the readability, cleanliness and elegance of expression - without incurring a performance penalty by using very large temporary values.

6.5 A Dynamic Stripmining Library

It is, however, possible for CENTORI to implement the stripmining techniques automatically in a library which implements the interface described in Chapter 5. To successfully use the technique, the library must be able to identify which values are temporaries by having access to the entire tree of operations being performed before starting the evaluation. This cannot be done with the standard eager evaluation model, instead this model must be replaced with a lazy evaluation scheme which stores "how" to calculate the result, rather than the final result.

6.5.1 Lazy evaluation

The interface does not enforce that every mathematical operation has been completed by the time the function or subroutine call returns, only that when the data is accessed by an Input/Output operation (e.g. a halo exchange) the result is the same as if it had been calculated

by an eager evaluation scheme. This flexibility allows the library to delay performing calculations until absolutely necessary i.e. allowing a lazy evaluation scheme to be implemented underneath the interface.

By only calculating the results when they are required, the full structure (i.e. AST) of the calculation becomes available to the library when it comes to evaluate the result. In the standard eager evaluation scheme the details of only a single operation are available, limiting the scope of potential optimisation. Calculations only occur when the results are absolutely required i.e. when calls are made to I/O operations defined in Section A.4, objects that are never used (values calculated and never written to the output or used in a subsequent calculation) are never evaluated, saving their computation. There is also scope to perform other optimisations such as substituting common combinations of operators with a single optimised operation, or performing factorisations and restructuring of operations which may offer an overall speed up.

6.5.2 Storing the method instead of the result

A lazy evaluation scheme requires the library to store the sequence of operations in the calculation rather than the running result. These "blueprints" have to be held in memory and must describe the calculation sufficiently so as to recreate the exact result when "replayed". There are at least two data structures which can be used for storing sequences of calculations to be replayed without ambiguity: trees and lists.

List evaluation

This method is a postfix operator scheme (similar to Reverse Polish Notation) where the calculation is held as a linear list of instructions (perhaps integers in an array) which when replayed manipulate a data stack[87]. Data is "pushed" onto the stack and "popped" off. Each operation pops its input arguments and pushes back its results. To add a new operation to the sequence the input lists are combined together and the additional operation added as the last instruction. This iterative scheme avoids the need for making recursive calls to a function and the expressions are completely unambiguous. However manipulating these structures in memory can be difficult as the relationship between individual operations is very difficult to maintain. Figure 6.9 shows how Equation 6.1 would be represented as a list of instructions and a data stack.

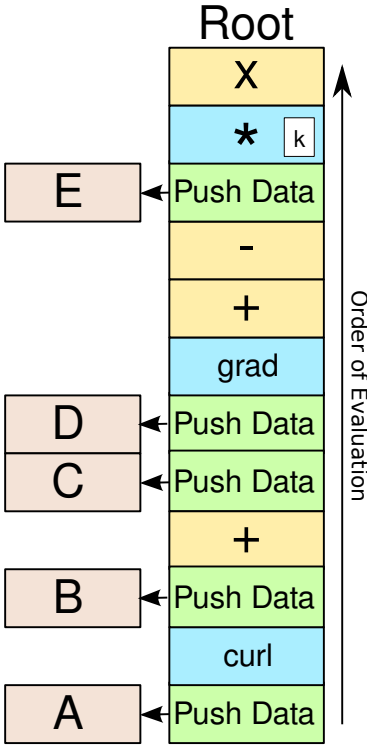


Figure 6.9: An example calculation stored in list notation.

$$(((\nabla \times \mathbf{A}) + \mathbf{B}) - (\mathbf{C} + \nabla D)) \times k\mathbf{E} \tag{6.1}$$

Tree evaluation

Alternatively, instead of storing the "blueprint" as a linear structure, the calculation is represented as a directed acyclic graph (DAG) or tree. Each node of the tree has a label with the operation to be performed and the input operands as children. The children can be another node in the tree or a childless node (leaf) which only stores an input data field with no associated operation. The root of the tree is the last operation to be performed and represents the result of the calculation. Performing a new function creates a new root node with input operands as its children[88].

To evaluate the result the tree is recursively descended until it reaches a childless node (leaf), the temporary is calculated and passed back up to the tree until each operation has been performed. Once the whole tree has been evaluated and the entire result is calculated and the intermediary temporary values are discarded. Figure 6.10 shows how Equation 6.1 would be represented as a list of instructions and a data stack.

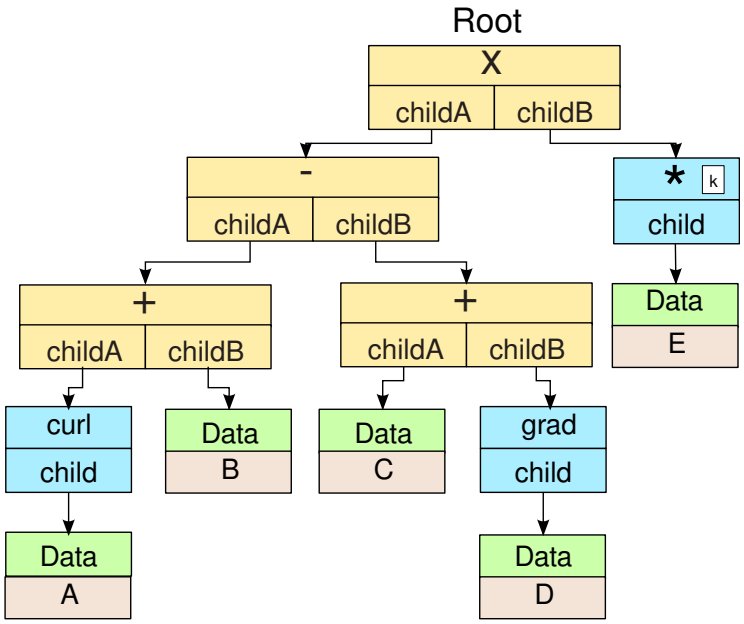


Figure 6.10: The same calculation as Figure 6.9 stored as a Directed Acyclic Graph.

6.5.3 When to evaluate

At some point the sequence of instructions has to be evaluated and the final result calculated. There are three circumstances that prompt this:

- Whenever data needs to be extracted from the interface; either from an Input/Output operation like halo exchange, the result has to be calculated before the operation can complete.

- When data types are copied between one another; rather than performing a deep copy of all the operations to be performed the field is evaluated and a shallow copy of the results is created.

- Once the tree reaches a certain size; the amount of temporary data required to evaluate the tree is recorded, when a tunable limit is reached the tree is evaluated. This prevents the tree growing too large that it would not fit in cache.

Whenever evaluation is required the library will iterate over the sequence of instructions for each block, calculating the results required from the calculation. It is also possible to enforce the evaluation using a call to the interface (`evaluate`). This performs no action in the case of eager schemes but causes a lazy evaluation scheme to calculate the value of a particular field. Using this is likely to inhibit performance and should be avoided wherever possible, only being

used where absolutely necessary or in debugging an implementation.

6.5.4 Identifying temporaries

When considering single isolated trees it is easy to identify which data is valuable and should be recorded in full and which is temporary data that is to be discarded when not needed. It is more complicated when performing real world calculations. In a physical application like CENTORI, many of the calculated values have physical meaning, these are labelled as separate values to help make the code clearer, however the values of the quantities themselves are rarely used. Consider

```
d = (a + b) + c
call evaluate(d)
```

This generates a small tree to be evaluated to calculate the value of `d` including a temporary value which stores the value of `(a + b)`. This is a true temporary value as it has no other name, it cannot be accessed after the evaluation and so can be "shrunk" to optimise the memory hierarchy. If this calculation were separated out like:

```
t = a + b
d = t + c
call evaluate(d)
```

This performs the same calculation in the same order, but gives temporary value a name, `t`. By explicitly labelling the temporary, it means it can be accessed after `d` has been evaluated. This means `t` loses its temporary status and must be stored in its entirety and cannot be "shrunk" . If `t` is an important quantity then storing the full field is an unavoidable but necessary measure, however if `t` were really just a temporary value giving it a name for longer than necessary can inhibit the performance of the application. It would be more efficient to use the first formulation if possible or to unassign the value from `t` before performing the evaluation, like this:

```
t = a + b
d = t + c
call unassign(t)
call evaluate(d)
```

Keeping track of how many names and references there are to a particular variable is easier when using a tree to record the instructions for lazy evaluation. It is also easier to keep track

of the number of references for memory management as described in Section 5.3.6. Once there are no references to a particular data object in any AST and the object does not have a name it has to be destroyed.

6.5.5 Self adaptive strip sizes

The size of the intermediate temporaries that form the "strips" in strip mining will affect how quickly the code takes to evaluate. As each AST is different, a single global strip size is likely to be suboptimal, some trees will have more nodes so may benefit from a smaller strip size, other will contain fewer nodes and may benefit from a larger strip size. Optimal global performance should occur when each evaluation tree uses the optimal local strip size.

This local minimum can be found automatically by measuring the performance of each AST with different strip sizes to search for the optimum performance. Repeating the same operation every time step is not an optimisation, instead the correct strip size needs to be easily retrieved and applied for each AST after it is found in the first instance. This can be done by "fingerprinting" the AST by generating a hash value for each node in the tree that is a function of the hash values of the node's children and the operation to be performed. The hash should uniquely identify the sequence of operations contained in the tree and can be used to identify one tree from another and the correct strip sizes associated with it.

This self-adaptive approach allows the real world performance to be fed back into the tunable parameters by making measurements on the code in the production environment. This allows the code to adapt quickly and automatically to changes in the architecture with only a small increase in the computation during the initialisation that is amortised over the length of the overall run.

6.5.6 Operator fusing

As the Abstract Syntax Tree is available to the library in a lazy evaluation scheme it is also possible to perform optimisations that are usually only available through static analysis or to the compiler. Fusing operations together in the same loop may increase the overall performance and if these occur commonly in CENTORI it would be beneficial to introduce them. These could be introduced by extending the interface but in many cases they do not fit with the standard Fortran notation. For example scale and add:

```
c = k*a + b
```

This is a common operation which would be initially implemented by two functions, with a temporary.

```
tmp = scale(a,k)
c = add(tmp, b)
```

Even with the dynamic cache strip mining it is usually more efficient to combine this into a new ternary operation which is not supported by the native `+,*,/,-` notation.

```
function scaleadd(c, a, b, k)
  do i=1,n
    c(i) = k*a(i) + b(i)
  end do
end function

call scaleadd(c, a, b, k)
```

Rather than extending the interface with this function and forcing every library implementation to include it, it may be preferable to only use it where it is advantageous. It would be relatively simple to substitute the two appropriate nodes in the evaluation tree with a single node that performed the single operation.

This contraction could be continued further, with a full scale data flow analysis and optimisation framework on the ASTs to be evaluated. This may lead to some major improvements in the performance of the code, however the analysis would be costly if performed every time an AST was evaluated. As the same trees are constructed at every time step a single analysis could be performed and stored using the same fingerprint method described in the previous section, however this would move the library into the domain of an optimising compiler which may be counter productive.

6.6 A Lazy Stripmining Library Implementation

Developing a library which performs lazy evaluation allows it to perform dynamic stripmining when it comes to evaluate the result. This has numerous advantages, as tracking the dependencies is complex and error prone for a human writing hand-tuned stripmining code. An

automatic implementation in the library would move all the book keeping onto the computer, a task it is well suited to, whilst keeping the "nice" features of the interface.

A third version of the library has been written to complement the reference version and the optimised version, which are both eager evaluation schemes. This version has the following features:

- Automatic construction of an Asymmetric Syntax Tree structure from the sequence of library function calls performed by CENTORI. This tree structure contains all the dependency and reference counting information required to manage the library.
- Automatic evaluation of results when required, either when an I/O operation is required, when the tree reaches a set size limit, when a deep copy is required or when forced to by the programmer.
- Full integration with the parallel decomposition and communication sections of CENTORI.
- Automatic identification and treatment of intermediate temporary values during evaluation. The library shrinks temporaries to a predefined number of "strips" in the z dimension and automatically uses modulus addressing to track data in the intermediate temporaries.
- Automatic management of all the dependencies for the complete AST, including finite difference and nearest neighbour operations. Calculation in advance of the main evaluation of required dependencies allowing the algorithms to proceed with all data available for every strip processed.
- Automatic memory management allowing full dynamic creation and disposal of data types and unevaluated ASTs.
- Implemented using the "array" memory layout and the appropriate fastest kernels from the benchmarks in Chapter 4 for arrays sized $34 \times 34 \times 3$ on HECToR with the PGI compiler.

All this functionality is achieved in CENTORI using the same interface defined in Chapter 5 and can be directly plugged into CENTORI without any high level changes. The library has been fully tested using the integrated CENTORI test suite and produces the same results to within floating-point precision. Due to time restrictions and complications in implementation this version of the library was unable to implement the self adaptive strip size mechanism or perform operator fusing.

6.6.1 Comparing the Eager and Lazy evaluation schemes

The dynamic stripmining techniques are an attempt to move calculations from a lower level of the memory hierarchy (like main memory) into a higher one (like one of the processor's caches). This is expected to be most effective when there is a strong difference between performance of memory levels, i.e. when moving operations from operating on the main memory system to any one of the processor's caches. This occurs in a strong scaling regime (where the total number of grid points being simulated is kept constant and divided between different numbers of processors) when the processor counts are much smaller, and larger amounts of data are held on each processor.

In cases where there is only a small amount of data on the processor, the stripmining may also have the effect of moving operations from a lower level of cache (e.g. L2 or L3) onto a higher level (e.g. L1), however the difference in performance between these levels is not as large as between main memory and cache. In these cases the overheads of using the cache blocking method may mask the improved memory bandwidth. It is likely that a simpler eager evaluation scheme will perform better because even though temporary data is whole vector or scalar fields, they are not very large and are likely to be held in cache anyway.

It is therefore expected that a lazy stripmining scheme will perform best on larger problem sizes and the eager scheme on the smaller problem sizes. The exact data size at which an eager scheme overtakes a lazy scheme is difficult to predict so a series of experiments were performed and are presented in Chapter 7 to compare the performance of the original version of CENTORI with the new CENTORI using the optimised eager evaluation library and the lazy stripmining library.

Chapter 7

Evaluating the Optimised Libraries

Chapter 5 describes an interface to a set of library functions that can be used by CENTORI while Chapter 6 describes the technical design of two implementations of the library functions. The first implementation is an "Eager" evaluation scheme that directly implements each function call using the best performing kernels from Chapter 4. The second uses a "Lazy Strip-mining" evaluation scheme to perform dynamic strip mining which is designed to optimise use of a processor's memory hierarchy. Both use the "array" memory layout and are optimised for HECToR's quad core Opterons with the PGI compiler.

This chapter presents a series of tests to compare the performance of these two library implementations against the original source code. The new libraries use only three components in their vector representation, dynamically converting between representations as required, whereas the original code used a nine component vector plus an integer to flag the representation updating all representations pre-emptively.

7.1 Testing Serial Performance

All of the optimisations applied in the new libraries attempt to reduce the execution time of CENTORI on individual processors, not the time spent communicating with other processors. There should be some improvement in parallel performance due to the reduced amount of data sent in halos (only three components are sent as opposed to nine) though this is a secondary effect, with specific parallel optimisation dealt with in Chapter 8. To evaluate the performance of the libraries and their optimisations on CENTORI as full codes, tests were run on small numbers of processors to minimise the amount of time spent in parallel communication. These

Vol per core inc halo	Vector field size	Equivalent #cores
$10 \times 10 \times 10$	24 KB	4096
$18 \times 18 \times 18$	139 KB	512
$34 \times 34 \times 34$	943 KB	64

Table 7.1: The serial performance test environment. The equivalent number of cores is the number required to decompose the standard $128 \times 128 \times 128$ simulation so each core has the correct field size.

tests were set up so that each processor had the same local data volume as it would when more processors are used on a much larger simulation. While performing the simulation has very little scientifically application, CENTORI is deterministic and the execution pathway will be exactly the same as on a full production run (including halo data volumes etc).

Table 7.1 describes the data volumes on each processor which form part of a $2 \times 2 \times 2$ parallel decomposition. This was required to occupy each processor socket completely as some resources, like the main memory bus, are shared between the multiple cores in the socket in shared memory systems. Running on a single processor would have left the other cores idle and would be an unrealistic portrayal of the real world deployment of CENTORI.

Testing was done on HECToR and HPC-FF using a version of CENTORI that was instrumented with internal lightweight timers which measure the wall clock time for a time step without introducing any measurable overhead in the application. Other measurements were from HECToR only, using the Performance Application Programming Interface[89] through the proprietary Cray Performance Analysis Toolkit which provides access to the hardware diagnostic counters in the Opteron processors.

Hardware performance counters provide information about the very low level operation of the processors which is usually unavailable. They provide figures relating to the cache usage, floating-point performance that cannot be acquired by any other means. Instrumenting the executable to record these counters does have consequences for the overall run time, increases in wall clock run time of 10% are expected. These changes are inevitable and though they distort the absolute value of some measurements (like timings and rates) the values are still useful for comparing between the different libraries and "strip sizes" tests.

7.1.1 Wall clock time step

The results of the measured wall clock time are ultimately the only performance metric that matters. Figure 7.1 shows the average wall clock time (over 1000 time steps) to complete one

Vol per core inc halo	Original	Eager	Minimum Lazy
$10 \times 10 \times 10$	32.1	24.9 (1.29)	29.8 (1.08)
$18 \times 18 \times 18$	186	104 (1.79)	112 (1.66)
$34 \times 34 \times 34$	1380	972 (1.42)	782 (1.76)

(a) HECToR

Vol per core inc halo	Original	Eager	Minimum Lazy
$10 \times 10 \times 10$	10.8	5.84 (1.84)	7.12 (1.51)
$18 \times 18 \times 18$	74.3	30.2 (2.46)	31.8 (2.34)
$34 \times 34 \times 34$	635	265 (2.40)	227 (2.80)

(b) HPC-FF

Table 7.2: Average wall clock time per time step on HECToR and HPC-FF in ms (Lazy value presented as the minimum recorded for any strip size). The value in brackets for Eager and Lazy schemes is the speed up factor over the original code.

model time step measured using the lightweight internal timers. Because the impact of these timings is minimal it is reasonable to expect the same levels of performance in a production run of CENTORI (of the same resolution and decomposition). Table 7.2a reports the actual recorded values (in milliseconds), using the minimum value for any strip size in the Lazy scheme and reporting the overall speedup of the code over the original.

The results confirm that where the data volumes on each processor are larger (e.g. 943 KB for the $34 \times 34 \times 34$ case), the Lazy scheme is more efficient than the Eager scheme. Though the schemes both perform much better than the original code on both platforms, the increased performance of the Lazy scheme over the Eager scheme is attributable to this better use of the memory hierarchy.

Where the local processor grid sizes are much smaller and the amount of data in a local vector field is far less ($10 \times 10 \times 10 = 23.4$ KB) the competition for space on the caches is reduced. In this situation, the additional complexity of the lazy strip mining scheme compared to the simpler eager scheme becomes a hindrance and the overall performance suffers. It might be expected that the lazy strip mining scheme would make better use of the higher levels of cache (e.g. L1) compared to the eager scheme in this regime, however it appears that the increased performance is so small as to be swamped by other factors.

From these results it can be concluded that the eager scheme should be used for large numbers of processors (≥ 512) with small amounts of data per processor and the lazy strip mining scheme in cases where there is a larger amount of data on each processor. On other architectures the same transition is very likely to occur though the exact position depends on the architecture's individual performance profile.

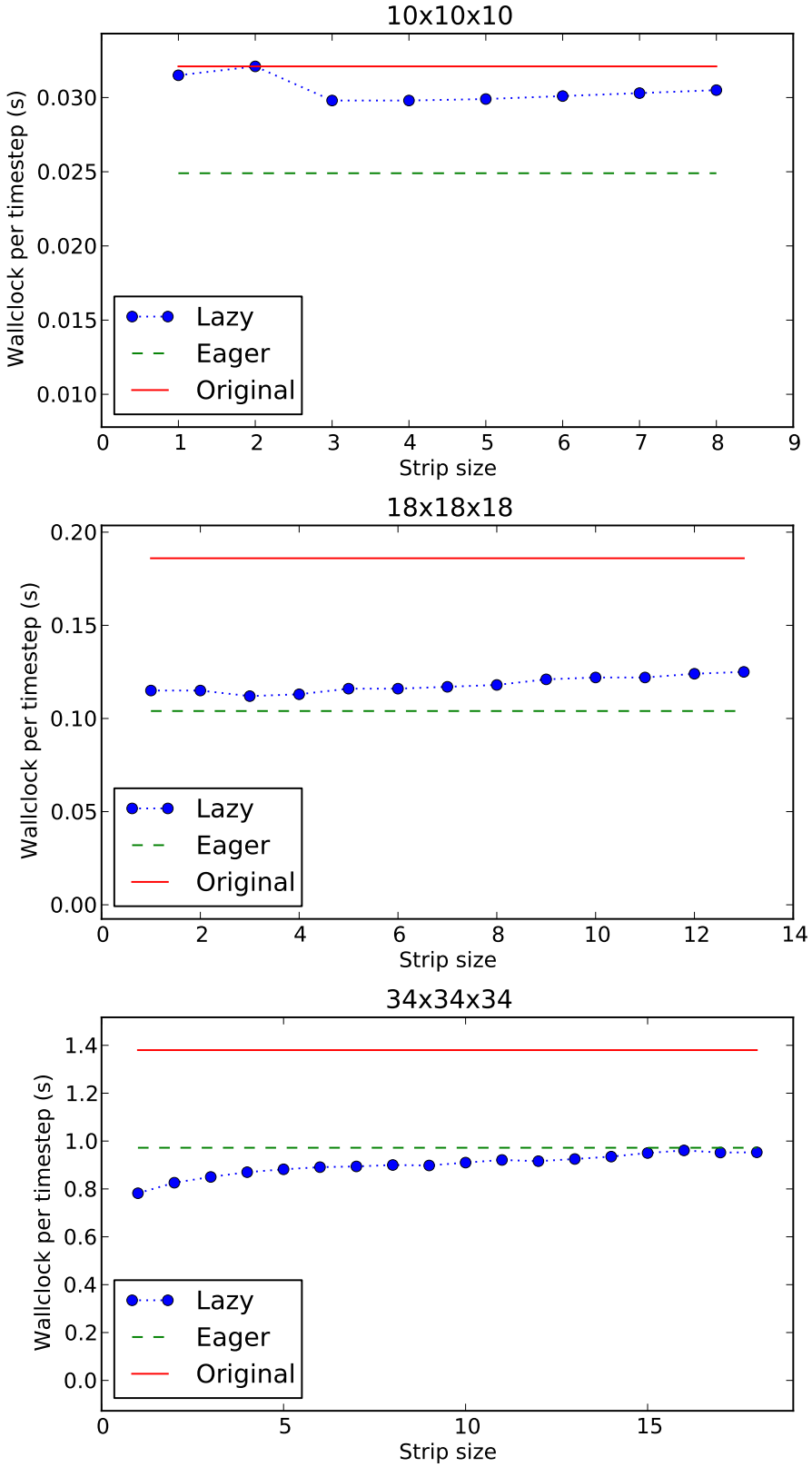


Figure 7.1: Average wall clock time per time step on HECToR (lower values indicate better performance).

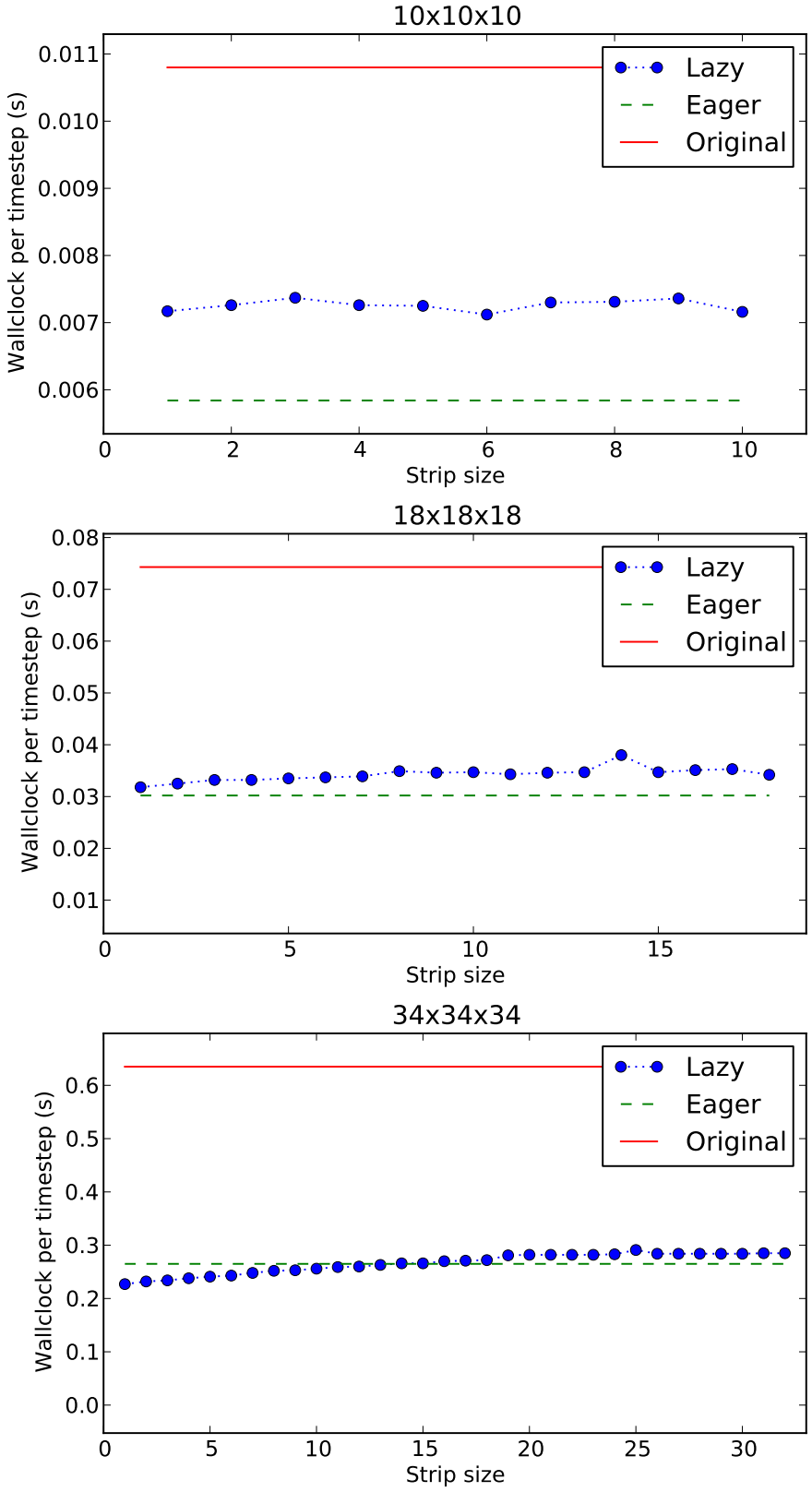


Figure 7.2: Average wall clock time per time step on HPC-FF (lower values indicate better performance).

7.1.2 Evidence based analysis

The previous section has shown that the overall run time has improved over the original code when using either of the two schemes, the features of the results fit with the hypothesis put forward for the changes in performance, however the single measurement of wall clock time per time step offers little evidence to reinforce the reasoning for the improvements. The new libraries incorporate many optimisations which address many different features of the target architectures and so further analysis is required to validate the reasoning. The PAPI and CrayPAT tools on HECToR can provide this.

7.1.3 Cache Optimisations

The Lazy scheme should improve the reuse of data on the cache, which in turn results in improved performance of the overall code. The "D1+D2 cache utilisation" measurement from PAPI and CrayPAT is a measure of the average number of times a cache line/block is accessed in either the Level 1 or Level 2 caches before it is reused. As a cache line is 64 Bytes long on the Opteron it contains 8 floating-point values so it should be expected that there be at least 8 references to a cache block before it is evicted if every item in the block is used once. Figure 7.3 shows the measured values for each of the library versions; features of note are:

- The original code only has a reference rate above 8 when operating on the smallest data volume ($10 \times 10 \times 10$), this is evidence to confirm the theory that the original data format of 9 component data types does poison the cache and reduces its usefulness. Similarly, when operating on the largest data volume, the Eager scheme has very few references per miss which suggest that its cache usage is quite poor. This was expected because of the larger data volumes will generate large intermediate temporaries that will flush the caches.

- The Lazy scheme is very cache friendly, increasing the number of times data in the cache is used before it is evicted significantly in all data volumes. The smaller the strip the greater the cache reuse, which is expected as the strip size is a direct measure of the size of the temporaries in use.

This evidence shows that the performance increase seen in the Lazy Evaluation scheme is attributable, at least in part, to improvements in the code's use of the memory hierarchy.

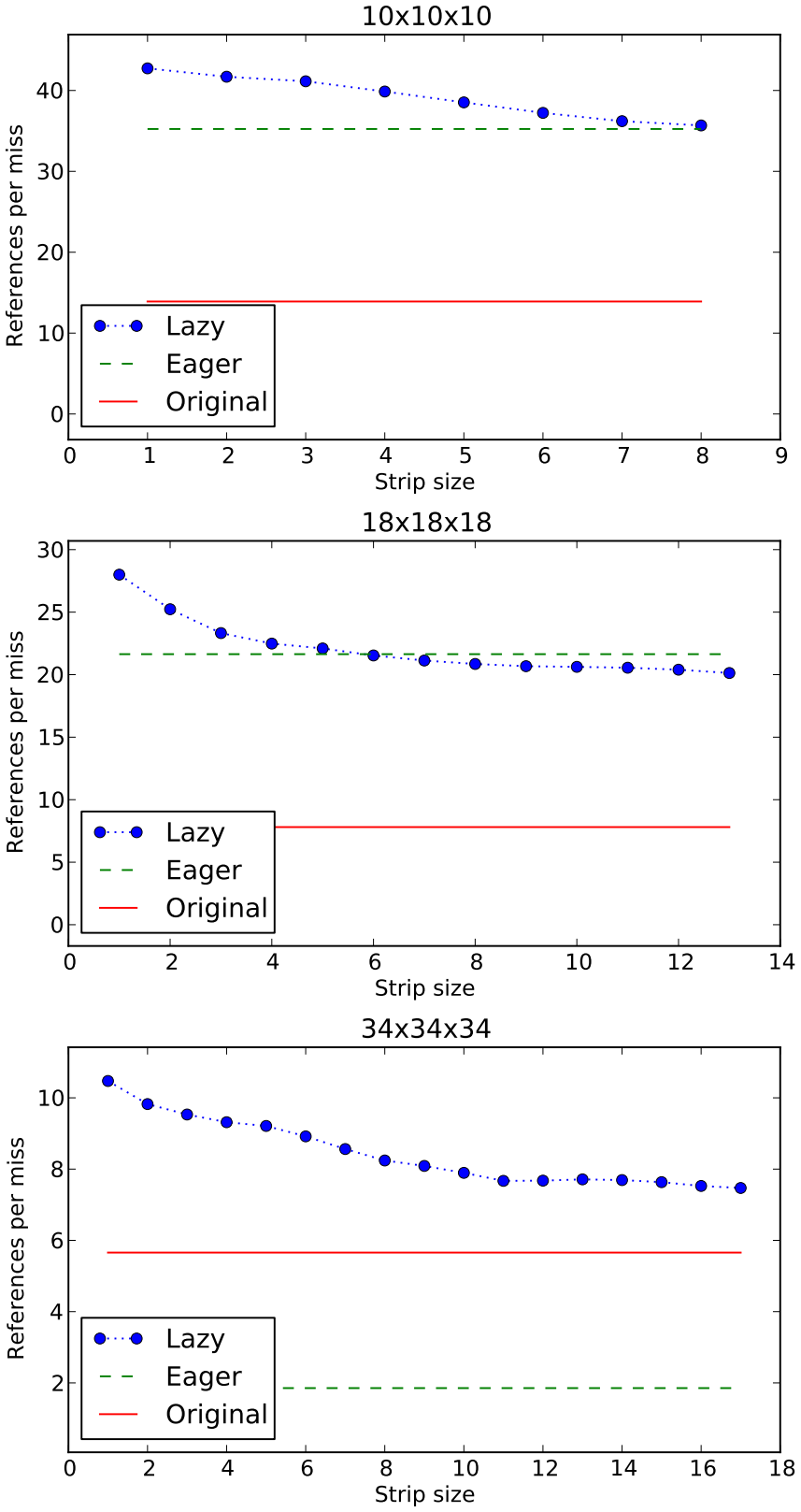


Figure 7.3: D1+D2 cache utilisation metric on HECToR. This reports the average number of times an element was accessed while it was in either the Level 1 or Level 2 cache of the Opteron.

Vol per core inc halo	Original Scheme	Eager Scheme	Lazy Scheme
$10 \times 10 \times 10$	3071	2955 (-3.8%)	2980 (-3.0%)
$18 \times 18 \times 18$	3451	3213 (-6.9%)	3239 (-6.1%)
$34 \times 34 \times 34$	3711	3395 (-8.5%)	3422 (-7.8%)

Table 7.3: The average number of floating-point operations performed per grid point, per time step. The figure in brackets is the percent change of the eager and lazy schemes over the original code.

7.1.4 Reduced number of floating-point operations

One of the originally discussed advantages of the change in memory layout from a nine components to only three was a reduction in the number of floating-point operations performed when updating each representation of the vector after one representation had been changed. This was balanced by the potential that there would be an increased number of calculations required to keep converting vector fields which do not change into different representations. It was posited that the number of floating-point operations could be reduced by the Lazy scheme by not evaluating unused branches of execution. Because code is only evaluated when it is required, if it is never referenced it will never be executed, however no such dead ends were detected in the code during the rewrite and so the overall improvement is expected to be minimal.

Table 7.3 shows the number of floating-point operations performed on average per model grid point per time step. This number is not constant for each of the grid sizes, which is evidence that the code does not scale linearly with the data volume which may be a feature for further investigation. It does, however, demonstrate there is an overall reduction in the number of floating-point operations performed in the new libraries over the original code, though there is no reduction in the Lazy scheme over the eager scheme. The improvements vary between 3.0% and 8.5%, which confirm there are parts of the code which do not scale linearly with data volume.

7.1.5 Improved ratio of SIMD instructions

Another advantage of moving from the nine component vector to the three vector array format was the ability to write code that could be easily vectorised by the compiler to issue SIMD instructions. It is an increasingly important feature of modern processors that they can execute multiple floating-point operations simultaneously by using Single Instruction Multiple Data resulting in a much higher overall floating-point performance (a factor 2 speed up on the Opteron and Nehalem processors). Table 7.4 shows the ratio of SSE (Streaming SIMD Extension)

Vol per core inc halo	Original Scheme	Eager Scheme	Lazy Scheme
$10 \times 10 \times 10$	11.6% : 88.4%	65.9% : 34.1%	66.2% : 33.8%
$18 \times 18 \times 18$	12.8% : 87.2%	61.5% : 38.5%	62.0% : 38.0%
$34 \times 34 \times 34$	14.2% : 85.8%	60.9% : 39.1%	61.2% : 38.8%

Table 7.4: This table shows the ratio of SIMD add and multiply instructions versus non-SIMD add and multiply instructions in each experiment.

add and multiply instructions issued by each library versus the number of equivalent scalar instructions. This was calculated using two of the metrics from the hardware performance counters which count the total number of floating-point operations performed and the number of SIMD instructions issued.

In the original version only 11% to 14% of instructions were SIMD whereas a significantly larger percentage (61% to 66%) are SIMD with the new data layout. If the code was totally bound by the throughput of the floating-point units, this shift in the balance between scalar and SIMD instructions would translate to a 33% to 40% reduction in run time. This represents a significant success in adapting the code for vectorisation, though there are still significant sections that still need attention or cannot be vectorised due to the algorithm (like the tridiagonal solvers).

To estimate the impact that SSE instructions have on the overall performance a separate version of the code was compiled with SSE instructions removed and tested against the original. The results are presented in Table 7.5, with figures comparing the "speed up" against the code including SSE instructions.

The results show that the SSEs appear to have the greatest impact on the Eager scheme on the smaller problem sizes, showing a consistent slow down to 85% of run time without them. It has been shown in Section 3.4.2 that SSEs only enhance performance when operating out of the highest cache levels, so it is likely that the improved performance of the Eager scheme with SSE instructions correlates with data mainly being in cache on these problem sizes. Interestingly the SSEs are so important to the Eager scheme's performance that removing them reduces its performance to equal or below the Lazy Strip mining scheme, making this scheme the best performing overall. On the largest problem size, which is operating out of main memory there is no difference in performance (in fact a 1% speed up).

In the original code the lack of SSE and prefetch instructions does not change the performance very much, slowing it slightly on the smaller problems but showing no difference on the larger problem size. Similarly there is very little variation in the Lazy Stripmining scheme,

Vol per core inc halo	Original	Eager	Minimum Lazy
$10 \times 10 \times 10$	32.8 (0.98)	29.8 (0.84)	29.8 (1.00)
$18 \times 18 \times 18$	188 (0.99)	123 (0.85)	114 (0.98)
$34 \times 34 \times 34$	1380 (1.00)	967 (1.01)	788 (0.99)

Table 7.5: Average wall clock time per time step on HECToR in ms for runs without any SSE instructions. Figure in brackets is the speed up (values less than 1 represent a slow down) relative to code that includes SSE instructions.

Vol per core inc halo	Original Scheme	Eager Scheme	Lazy Scheme
$10 \times 10 \times 10$	0.985	0.436 (44.2%)	0.436 (44.2%)
$18 \times 18 \times 18$	3.43	1.40 (40.8%)	1.40 (40.8%)
$34 \times 34 \times 34$	12.8	5.00 (39.0%)	5.00 (39.0%)

Table 7.6: Total volume of data transferred using MPI per time step in MB. Figure in brackets for Eager and Lazy scheme is a percentage compared the volume transferred by the original scheme.

despite the increase in the proportion of vector instructions. This suggests the extra accounting and book keeping involved in the lazy strip mining scheme is preventing the code from taking advantage of the additional floating-point performance.

7.1.6 MPI Data transfer volumes

By changing the data layout from nine components to three, vector fields occupy less than one third of the space that they had done previously. The amount of data transferred between processors correspondingly decreases as boundary exchanges for halos would be required to transmit only one third of the data they had originally. Table 7.6 records the amount of data transferred between all processors on average per time step. Although the new data layout scheme translates to a significant reduction in the amount of data being transferred over the network, there is not a total reduction to one-third as the change in data layout does not affect scalar fields and other forms of parallel communication. Records show that there are on average 2443 messages transmitted during a time step which translates to an average message size of the order of KB for most messages. At this level the latency of the communication will dominate its transmission time. Section 3.1.2 show that latency on a 1KB messages is almost 100% of transmission time on HPC-FF and approximately 50% on HECToR and HPCx from the results in . This means the reduction of data volume has only a limited effect on improving performance.

Vol per core inc halo	Original Scheme	Eager Scheme	Lazy Scheme
$10 \times 10 \times 10$	95.7 (1.04%)	118.7 (1.29%)	100.0 (1.09%)
$18 \times 18 \times 18$	108.2 (1.18%)	180.2 (1.96%)	168.6 (1.83%)
$34 \times 34 \times 34$	105.7 (1.15%)	143.7 (1.56%)	180.1 (1.96%)

Table 7.7: Floating-Point operation rate in millions of floating-point operations per second (MFLOPs). The figure in brackets is the % of peak double precision performance on HECToR.

7.2 Performance Analysis

Many attempts at optimisation aim to reduce the load on the floating-point unit which is considered the bottleneck to improved performance. However Table 7.7 shows the average rate of floating-point instructions processed as a percentage of the theoretical peak floating performance achievable by the processors. This measurement is for the code as a whole and so includes all the time spent waiting on parallel operations and other non-floating-point dominated sections.

The values of between 1.0% to 2.0% of peak performance show that the rate the floating-point unit operates at is not the limiting factor of CENTORI's performance. Instead the rate at which data arrives to the processor, the memory bandwidth dominates. This justifies work on the Lazy scheme which improves cache re-use which improves the effective memory bandwidth and the move from nine component to three component vector data types in both the Eager and Lazy schemes. This is reflected in the improved wall-clock performance of CENTORI on both HECToR and HPC-FF measured in Section 7.1.1.

Chapter 8

Evaluating and Optimising CENTORI's Parallel Performance

Chapter 7 documents the change in the serial performance of CENTORI after applying the Eager and Lazy-Strip mining schemes. These show significant improvement in the performance of the code, however CENTORI is a parallel code. Unless there is a similar gain in parallel performance, the scalability of the code will suffer and the serial gains will not be realised.

8.1 Quantifying CENTORI's Performance

Figure 8.1, and Tables 8.1 and 8.2, detail the results of profiling the original version of CENTORI using its lightweight internal timers. The time is the accumulated time on all processors on each of the designated tasks:

– *Halo Exchange* - Covers the time spent copying data to and from the parallel buffers and time spent waiting for MPI send and receive operations to complete during all halo exchange operations.

– *Integration* - Covers all the time spent performing integrations along the ψ dimension, including a call to `MPI_Scan` and time spent calculating the integral value on individual processors.

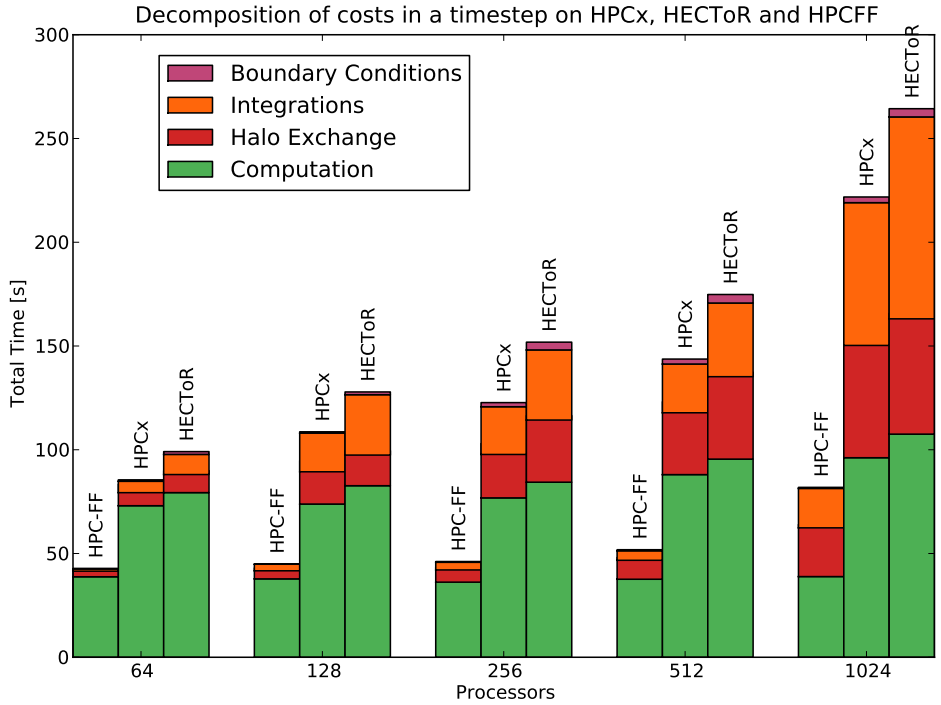


Figure 8.1: CENTORI time step costs - A decomposition of the time spent by all processors in each part of CENTORI's time step.

Activity	HPCx	HECToR	HPC-FF
Boundary Conditions	0.7 (0.8%)	1.3 (1.3%)	0.2 (0.3%)
Integration	5.5 (6.1%)	9.7 (9.6%)	1.1 (2.6%)
Halo Exchange	10.4 (11.6%)	10.6 (10.5%)	3.9 (8.9%)
Compute	73.0 (81.5%)	79.4 (78.6%)	38.8 (88.2%)
Total	89.6	101.0	44.0

Table 8.1: The decompositions of the cost of a time step on 64 processors on HPCx, HECToR and HPC-FF with percentages of total wall clock time in brackets.

Activity	HPCx	HECToR	HPC-FF
Boundary Conditions	2.7 (1.2%)	4.0 (1.5%)	0.6 (0.7%)
Integration	68.8 (29.9%)	97.3 (36.4%)	18.8 (22.6%)
Halo Exchange	62.3 (27.1%)	58.2 (21.8%)	24.1 (28.9%)
Compute	96.2 (41.9%)	107.5 (40.3%)	39.8 (47.8%)
Total	230.0	267.0	83.3

Table 8.2: The decompositions of a cost of a time step on 1024 processors on HPCx, HECToR and HPC-FF with percentages of total wall clock time in brackets.

– *Boundary conditions* - Covers time spent computing boundary conditions on axis and edge processors and includes time spent in `MPI_Allreduce` when calculating averages for the axis point.

– *Computation* - All the remaining time spent by processors in the time step.

The figures show that computation sections grow in cost by 33% between 64 and 1024 processors on HPCx and HECToR. This can be attributed to the increased amount of computation caused by the increasing amount of data held in halos. Halos sizes are proportional to the surface area of all processors so the more processors the problem is decomposed between, the greater the amount of repeated data held in halos and the increased amount of additional computation.

Meanwhile the cost of the parallel tasks increases more significantly: Boundary Conditions increase by factors between 5.5 and 6.2 times, Halo Exchanges between 3.0 and 3.9 times and Integration by factors between 10 and 17 times. The time spent in the parallel sections increases from between 10% and 20% of the total cost on 64 processors to between 50% and 60% at 1024 processors. This represents a significant increase in the total time spent doing the secondary work of communication rather than the primary work of computation. This inhibits the scaling of the code as shown in Figure 8.2 which shows how CENTORI's scaling begins to falter after 1024 processors on all platforms.

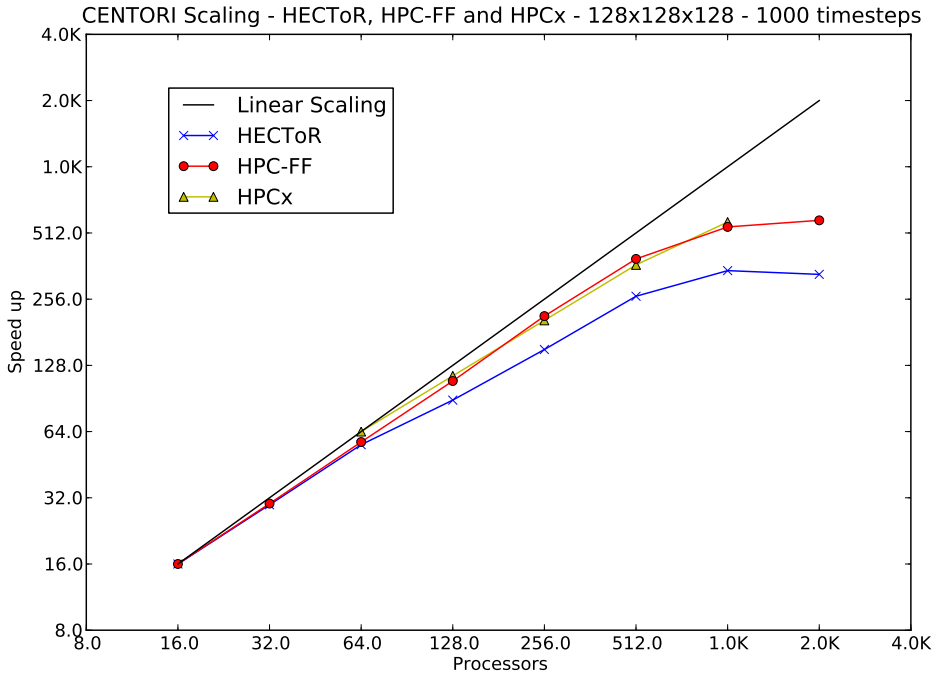


Figure 8.2: CENTORI's scaling on HECToR, HPCx and HPC-FF (speed up is vs 16 processors).

8.2 Addressing CENTORI's Parallel Performance

The previous chapters have demonstrated a significant improvement in the performance of the Computation sections in CENTORI. These improvements will cause an overall speed up but do not address the significant cost of the parallel code at high processor counts, which must be addressed to maintain the balance between communication and computation in improved versions of CENTORI. Several changes to the parallel scheme follow which should reduce the cost of CENTORI's communication.

8.2.1 Reducing Data Volumes

Though moving from nine component vectors to three components was an optimisation primarily focused on improving CENTORI's serial performance it should help improve CENTORI's overall parallel performance as well. Section 7.1.6 showed that the total volume of data transmitted over the network using MPI was reduced by between 2.0 and 2.4 times. This change in data volume reduces the time spent in the Halo exchanges where the largest data transfers take place, however the reduction operations in CENTORI are based on much smaller data volumes and so are dominated by the latency of the network. Changing the number of components in a

Example 8.1 A repeated call to a reduction operation in CENTORI which could be aggregated.

```
do i=1,n
  ...
  call MPI_Allreduce(input(i), output(i), 1, &
          MPI_INTEGER, MPI_SUM, comm, err)
  ...
end do
```

vector object will have little to no effect on these operations.

8.2.2 Pre-posting Receive Buffers

Though many hardware implementations provide a universal buffer for data to be stored in if it occurs unexpectedly (e.g. if a send operation occurs before the corresponding receive operation is posted) it is suboptimal to use them as they are of limited size and using them causes an additional copy of the data. A better method is to ensure that the receive buffers are available as early as possible so the received data can be copied directly into the correct location. This method is more tolerant of any small discrepancies in processor synchronisation and reduces the amount of time that each processor is waiting for its neighbours to transmit and receive data.

8.2.3 Message aggregation

With most global communication, the amount of data being transmitted is quite small, of the order of 10 - 100s of bytes, so it is the latency of the link which limits the time to completion rather than the bandwidth. In the original CENTORI many reduction operations were performed inside loops which called the same reduction operation multiple times with different data, waiting each time for the last to complete before starting the next. This incurs a heavy penalty as each operation will have been delayed by the latency whereas by aggregating the repeated reductions into a single larger reduction which operates on a larger data volume the latency penalty is incurred only once. Also, because the data volumes are so small, the overall cost of the communication can be significantly reduced.

This form of message aggregation was applied to the reduction operations which are core parts of the Integration and Boundary conditions sections of CENTORI. In most cases it has been possible to replace N operations on 1 double precision floating-point value with a single

Example 8.2 A revised version of Example 8.1 which combines multiple reductions into a single call.

```
do i=1,n
  ...
end do

call MPI_Allreduce(input(:), output(:), n, &
                   MPI_INTEGER, MPI_SUM, comm, err)

do j=1,n
  ...
end do
```

Activity	HPCx	HECToR	HPC-FF
Boundary Conditions	2.0/0.5 (4.3)	3.7/0.5 (7.5)	0.4/0.2 (1.9)
Integration	23.0/1.2 (18.7)	33.8/1.1 (29.6)	3.7/0.5 (6.8)
Halo Exchange	26.2/7.5 (3.5)	32.1/9.6 (3.4)	7.3/3.5 (2.1)
Compute	76.8/33.2 (2.3)	84.4/47.9 (1.8)	36.2/17.3 (2.1)
Total	128.0/42.3 (3.0)	154.0/59.1 (2.6)	47.5/21.5 (2.2)

Table 8.3: Measured cost of each task in seconds when running on 256 processors - Old/Eager (Speed up).

operation on N double precision floating-point values. Examples 8.1 and 8.2 show the difference.

8.3 Evaluating CENTORI's Performance

After making the changes documented in the previous section as well as including all of the overall performance enhancements documented in Chapter 7 to implement the Eager evaluation scheme it is possible to evaluate the overall effect of the performance improvements.

8.3.1 General Improvement

Figure 8.3 show the equivalent measured values for the improved CENTORI as described in Section 8.1. It depicts the components when running with the improvements to the parallel code and with the Eager serial improvement code described in previous chapters. The very large processor decompositions seem to show large growth in the cost of computation on HPC-FF which is as yet unexplained on this platform. Table 8.3 contains the cost of each component on 256 processors on each platform for the original CENTORI (first value) and for the improved CENTORI (second value), and the speed up (in brackets).

These figures, along with the results in the graph, show that the greatest improvement is in

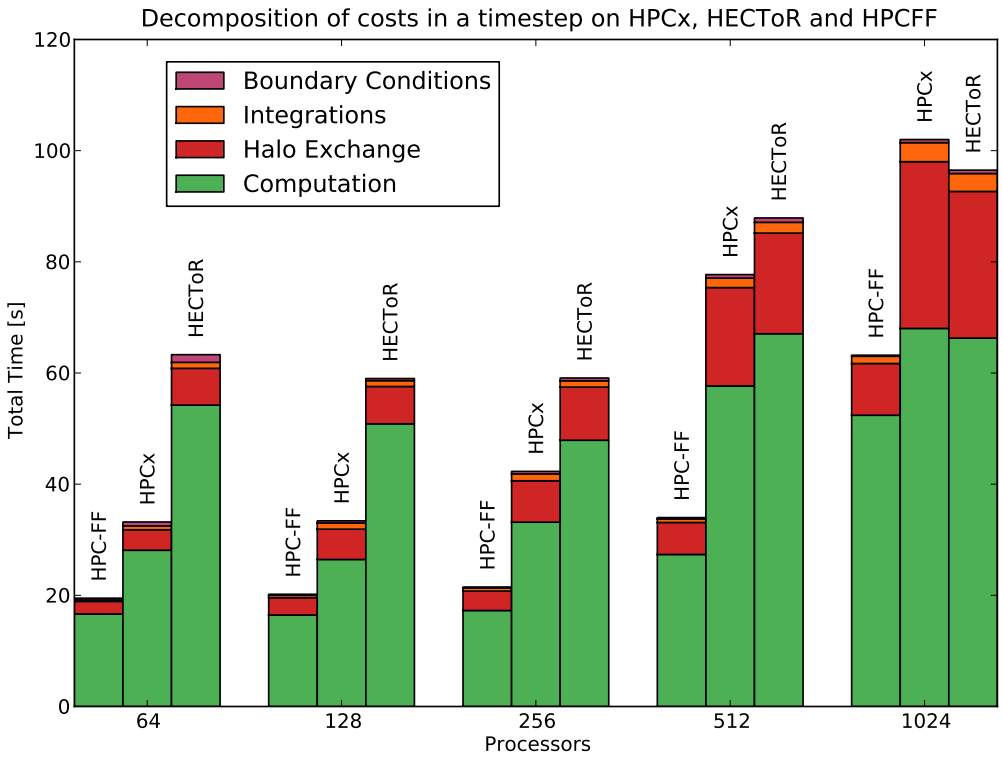


Figure 8.3: Decomposition of time step costs in CENTORI using the Eager serial scheme and the parallel improvement such as message aggregation.

the time spent on integrations, with the biggest speedup on platforms with the highest latency in their networks (HPCx and HECToR). The improvements are so significant that Integration is no longer has a significant impact on the overall performance of code and is a similar order of magnitude to the boundary condition code.

Similarly, the Halo Exchanges are now approximately 3.5 times faster on HECToR and HPCx and over twice as fast on HPC-FF. This effect is a combination of the reduced data volume and the improved structuring of the Receive commands. Compute times show a speed up of between 1.8 and 2.3 times which are similar to the figures seen in Chapter 7.

The overall speed up is also significant, showing an overall 3.0 times speed up on HPCx where there were significant issues with parallel and serial implementations, and a 2.2 times speed up on HPC-FF where the parallel performance was determined to be much better than HPCx's or HECToR's.

8.3.2 Scaling

Figure 8.5 shows how CENTORI performs during strong scaling with a constant problem size. Each graph shows the scaling of the Original version and the scaling of the Eager serial scheme with the parallel modifications. HECToR demonstrates super linear scaling between 128 and 256 processors, probably due to the problem being moved from main memory into the cache, and continues to scale to 1024 processors. The new code shows superior scaling over the original code on HPCx below 512 processors, but does appear to continue scaling up to 1024 processors. Scaling on HPC-FF seems to be much poorer with the new code, possibly as a result of exceptional performance of the code at 64 processors compared to higher processor counts. The performance also seems to be limited to 1024 processors.

These results may be misleading, as overall there is a general speedup on all processor counts between the Original code and the new versions of the code. Scaling is only a measurement of speed up versus a baseline performance, so it can give the false impression that a slower, but more scalable, code outperforms the faster code.

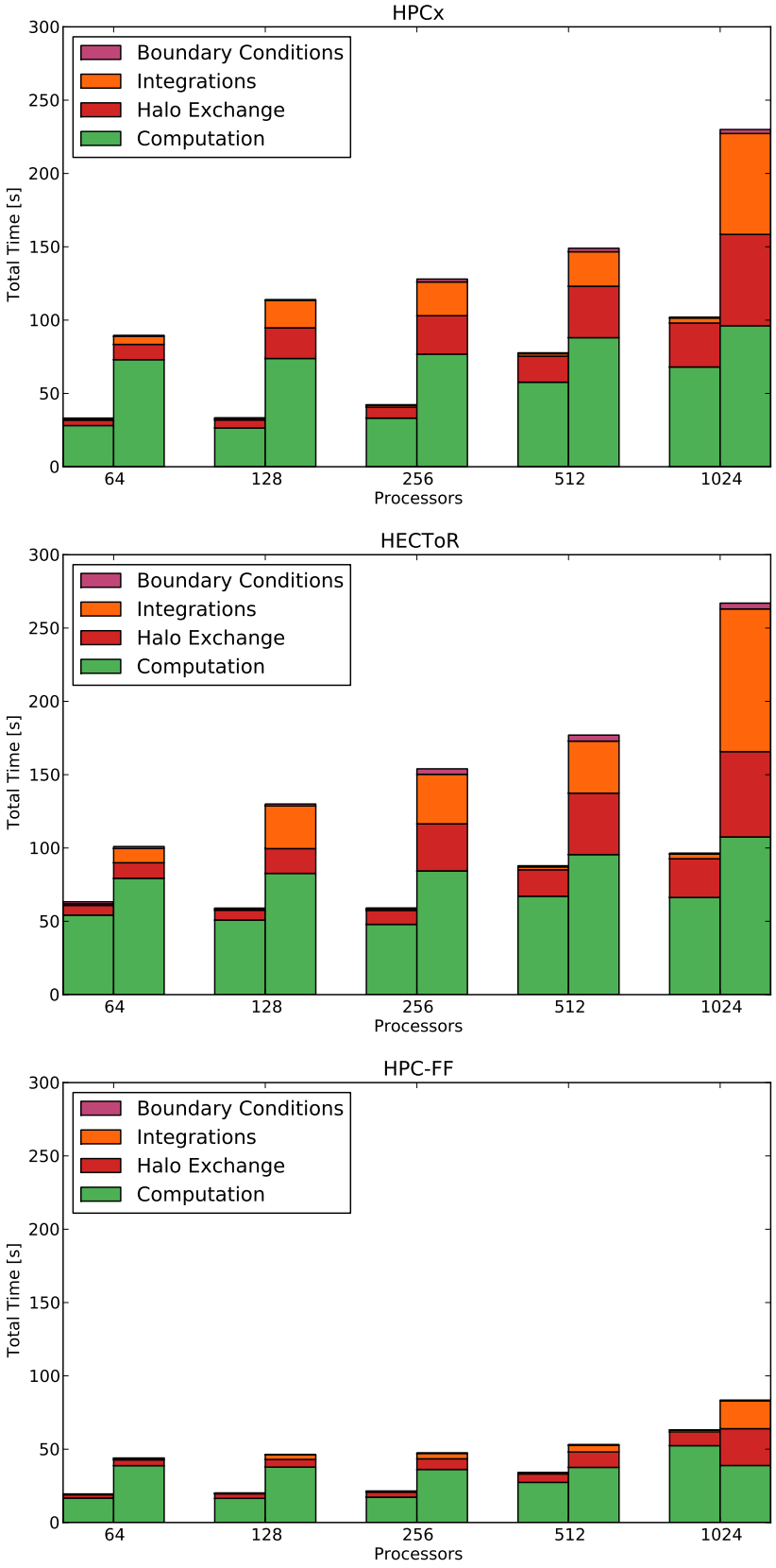


Figure 8.4: Comparative performance of the original CENTORI (right) and the Eager scheme (left) with additional parallel improvements for each architecture and number of processors.

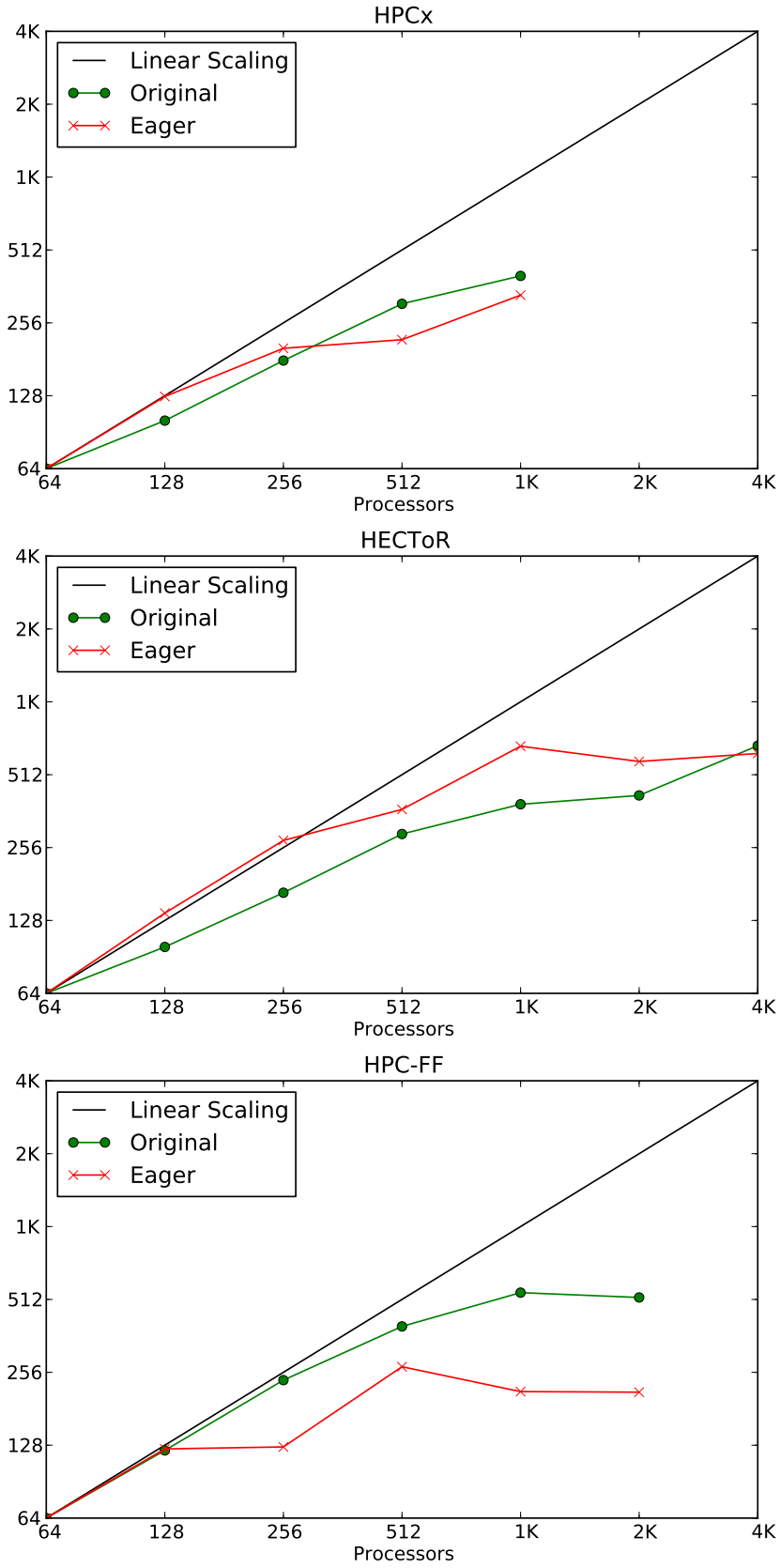


Figure 8.5: Scaling speed up against 64 processor for CENTORI on HPCx, HECToR and HPC-FF.

# ψ procs	# θ procs	# ζ procs
2	2	16
2	4	8
2	8	4
2	16	2
4	2	8
4	4	4
4	8	2
8	2	4
8	4	2
16	2	2

Table 8.4: Combinations of processors decompositions for 64 processors.

8.3.3 Processor Aspect Ratios

CENTORI has several restrictions on the nature of the aspect ratio of processor decompositions and the whole model grid size.

1. It can only operate on global resolutions which are a power of 2.
2. The number of processors in each direction must also be a power of 2.
3. There must be a minimum of 4 data points on each processor.

These restrictions mean the core data volumes, without halo data, are always a power of 2 and within the restrictions there are many potential processor decompositions. For example, there are 10 valid ways to decompose a $128 \times 128 \times 128$ resolution between 64 processors as shown in Table 8.4 and for any grid combination of 2^n processors there are $\sum_{i=1}^{n} i$ ways to decompose the problem.

But which of these decompositions produces the best result for CENTORI? All previous decompositions have attempted to minimise the amount of time spent by the parallel sections of the code. By keeping the data volumes as close to a cube as possible, the surface area (also the volume of data held in the halos) decreases and so less data is transferred. Table 8.5 shows the decompositions used to achieve the standard processors numbers in all standard scaling tests. Where it is impossible to create a cube, the number of processors in the ψ dimension has been increased first then the number of processors in the θ dimension. However the serial features of the code may perform better on different decompositions, e.g. those with longer ψ dimensions, due to the nature of the optimised code.

To identify the best decomposition of the old and the new models a large number of tests were performed covering all possible decompositions for processor counts between 64 and 1024.

Total Procs	# ψ procs	# θ procs	# ζ procs
64	4	4	4
128	8	4	4
256	8	8	4
512	8	8	8
1024	16	8	8
2048	16	16	8
4096	16	16	16

Table 8.5: Default processor decompositions used for standard scaling tests with CENTORI.

This produces a large number of data points for each processor configuration which are presented in Figure 8.6 as a box and whisker plot. The median value is represented by the solid horizontal line inside the box while the limits of the box itself are the first and third quartiles of the data distribution. The whiskers show the overall limits of the distribution, from the maximum cost to the minimum cost.

Tables 8.6 and 8.7 show the maximum and minimum values recorded and the decompositions that created them and also the standard result. Figure 8.6 shows there is a large difference between the maximum and minimum values recorded when operating at large processor counts, with the most costly decomposition costing over 7 times the cheapest in the optimised case.

1. For lower processor counts, the spread of the results is much smaller with the optimised code. The optimisations appear to reduce the impact that a poor decomposition has on the overall run time of the code.

2. With the original code, the fastest decompositions are up to 10% faster than the standard decompositions. In three of the five cases with the optimised code, the fastest case is less than 5% faster.

3. In the original case the fastest decompositions tend to have a larger number of processors in the ζ dimension. In the optimised code, the code appears to run fastest with a large number of processors in the ψ dimension. This is probably a function of the dominance of the "integrate" function across the ψ arrays in the original code.

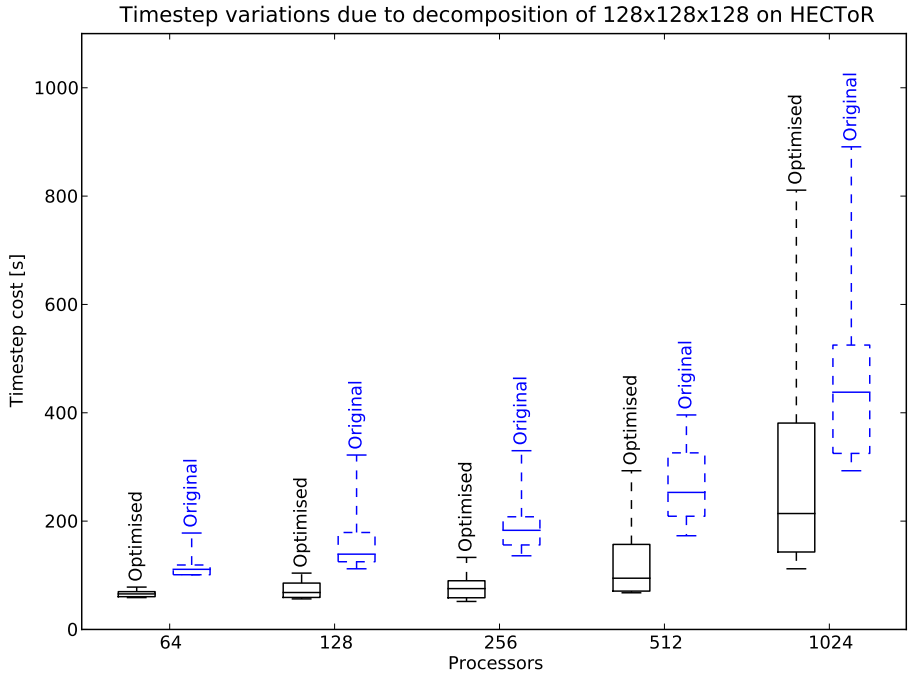


Figure 8.6: Box and whisker diagram showing the per range and average per time step cost for each processor count for the original code and the version including all the optimisations.

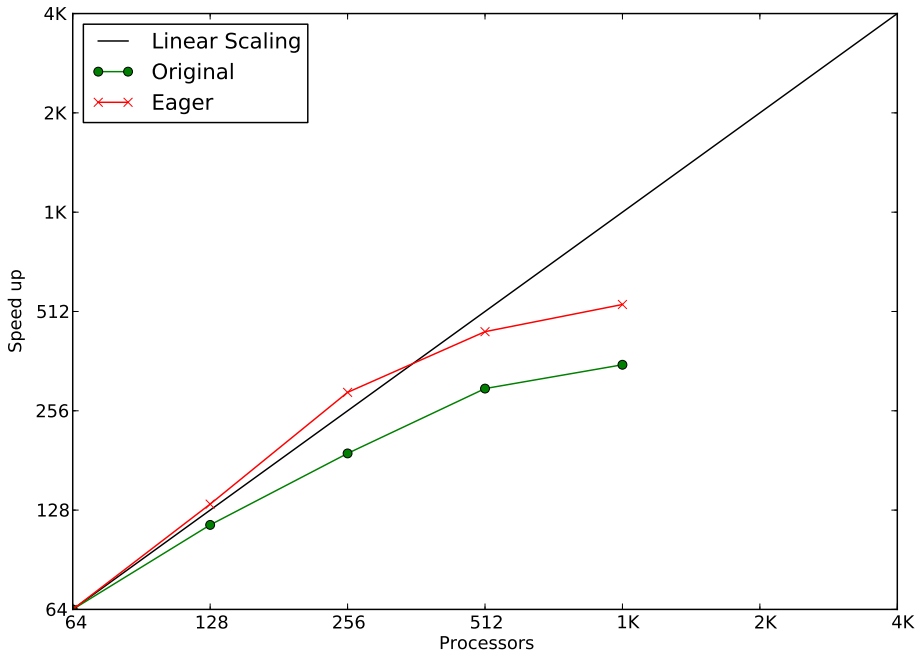


Figure 8.7: Scaling performance of CENTORI using the optimal aspect ratio for each processor count.

# Procs	Min	Max	Standard
64	100.5 ($2 \times 4 \times 8$)	177.3 ($16 \times 2 \times 2$)	101.1 ($4 \times 4 \times 4$)
128	112.3 ($4 \times 4 \times 8$)	321.3 ($32 \times 2 \times 2$)	129.3 ($8 \times 4 \times 4$)
256	136.2 ($4 \times 4 \times 16$)	330.2 ($32 \times 4 \times 2$)	156.9 ($8 \times 8 \times 4$)
512	173.6 ($8 \times 4 \times 16$)	396.3 ($32 \times 8 \times 2$)	184.8 ($8 \times 8 \times 8$)
1024	292.9 ($16 \times 4 \times 16$)	890.9 ($2 \times 32 \times 16$)	325.6 ($16 \times 8 \times 8$)

Table 8.6: Original CENTORI decomposition time step costs (wall clock seconds).

# Procs	Min	Max	Standard
64	61.9 ($8 \times 4 \times 2$)	82.2 ($2 \times 2 \times 16$)	63.6 ($4 \times 4 \times 4$)
128	59.3 ($8 \times 4 \times 4$)	109.7 ($2 \times 2 \times 32$)	59.4 ($8 \times 4 \times 4$)
256	54.4 ($16 \times 4 \times 4$)	140.1 ($2 \times 4 \times 32$)	60.1 ($8 \times 8 \times 4$)
512	71.1 ($32 \times 4 \times 4$)	308.8 ($2 \times 8 \times 32$)	71.1 ($8 \times 8 \times 8$)
1024	117.5 ($32 \times 16 \times 2$)	853.7 ($2 \times 32 \times 16$)	154.1 ($16 \times 8 \times 8$)

Table 8.7: Optimised CENTORI decomposition time step costs (wall clock seconds).

Overall, the processor layout can have a significant impact on performance, though usually the default combination is one of the more competitive results. Choosing a poor layout (i.e. the Max values in the previous table) can cause the code to run many times slower. When running long simulations with CENTORI it would be beneficial to identify the fastest processor layout for the architecture in question. Though these results are generally interesting they are architecture specific and further analysis on the other platforms may produce different results.

Chapter 9

Conclusions

Software represents such a significant investment of time and effort that it must transcend individual generations of hardware. In the HPC domain, however, it is not only functionality and maintainability but performance that is critical. Ideally applications are optimised to achieve the best performance on every system. The work discussed in this thesis achieves this goal in three ways:

1. Primarily and most pragmatically, it runs much faster on each of the target architectures. The improvements come from the significant optimisation of the serial instruction stream and data structures in memory, along with refinement of the parallel communication pattern. The balance between these two factors has been maintained, such that the code continues to scale to 1024 processors whilst running over 2.5 times faster on HECToR and HPCx.

2. The optimisations have been implemented in the code without obscuring, obfuscating or specialising the original source code. Instead, going against perceived wisdom, the scientific equations are represented in the code more clearly, are easier to manipulate yet are executed significantly quicker. This has been achieved by reversing the tight integration between optimised source code and CENTORI's fundamental algorithms by using an abstract interface.

3. The abstract interface approach significantly reduces the scope over which optimisations have to be applied. The abstraction is sufficiently general that optimisation strategies which would previously require numerous changes across the entire body of code are

contained within a single library implementation. The interface is sufficiently general that the application can perform runtime analysis and optimisation when evaluating abstract syntax trees. This technique directly improved CENTORI's performance when using stripmining and lazy evaluation on large problem sizes.

Individually, these modifications draw from a variety of domains within computer science; compiler theory, parallel programming, instruction stream optimisation and language design. Combining them in a single application, especially within the HPC domain, represents a significant and novel contribution to the field.

This approach aim provides CENTORI with an optimisation framework for the next decade. It allows a more productive interaction between physicists and optimisers that minimises the amount of of domain knowledge each requires of the other's field. Similarly, these techniques can be applied more widely than just CENTORI; any application that applies operators to large fields of data could derive similar benefit from these optimisation techniques. Many Computational Fluid Dynamic codes could adopt this approach with relatively limited changes to many of the data structures and operators.

9.1 The Abstraction

One of the most important contributions of this work was the strict separation of the implementation details from the algorithmic descriptions of the model through a novel interface described in Chapter 5. This forces the performance critical kernels and the plasma physics to become distinct parts of the code. The design of this interface has two major advantages:

– CENTORI's core fundamental equations are now expressed in a language that is very similar to vector algebra and vector calculus notation. This language makes the code far more readable and is very familiar to the core developers, who are primarily physicists. It allows them to quickly prototype ideas or further refine the model because the physical equations on paper are directly comparable to the source code.

– New memory layouts and optimised kernel modules can be "plugged in" to CENTORI at compile time without altering any other source code. As well as defining a library of optimised subroutines, they can define the internal structure of fundamental data types which allows them even greater freedom to optimise for performance. As the interface

	HECToR			HPC-FF			HPCx		
Procs	**Old**	**New**	×	**Old**	**New**	×	**Old**	**New**	×
64	1.578	0.989	(1.60)	0.688	0.305	(2.26)	1.400	0.519	(2.70)
128	1.016	0.461	(2.20)	0.362	0.158	(2.30)	0.891	0.261	(3.41)
256	0.602	0.231	(2.61)	0.186	0.084	(2.20)	0.500	0.165	(3.03)
512	0.346	0.172	(2.01)	0.104	0.066	(1.57)	0.291	0.152	(1.92)
1024	0.261	0.094	(2.77)	0.082	0.062	(1.32)	0.225	0.100	(2.26)

Table 9.1: Wallclock per time step (in seconds) for $128 \times 128 \times 128$ CENTORI runs on each system with the original code (Old) and the eager scheme (New) and relative improvement (×) averaged over 1000 time steps.

splits operations into individual functions, testing and verifying correctness can be done at the unit level which is far quicker and easier than when they are fully integrated.

9.2 Performance

Creating another level of abstraction, like the interface and library, could be detrimental to performance as each additional layer above the hardware tends to bloat the code and incur additional performance penalties. While there may be some overheads introduced by the new code, good performance can still be achieved which the two example library implementations demonstrate. One provides a standard "library of functions", like BLAS, using the kernel that has the best measured performance on HECToR in Chapter 4 for each function call. The other uses a similar set of optimised kernels, but relies on lazy evaluation and "stripmining" to automatically minimise traffic over the memory subsystem as described in Chapter 6. This innovative and novel approach caches more data and shows much better performance than the first library in regimes where the majority of data resides on main memory. Either approach can be selected at compile time and will produce the same results. Both approaches showed significant speed improvements over the original code on the current hardware as demonstrated by the figures in Table 9.1 for the eager scheme and the results in Chapter 7 for the lazy stripmining scheme.

The combination of a clear and concise interface to a high performance library which is also capable of performing sophisticated transformations and optimisations is a rare combination in a scientific application. While other large frameworks exist that allow scientists to express concepts in physical terms and use highly optimised kernels to perform them they tend to be for much larger projects (e.g. like the Chroma suite for Lattice-QCD research[90]).

Unfortunately, the project did not create full implementations for each of the memory layouts

described in Chapter 4. One of the primary extensions to the work would be to create library implementations based on the cyclical and/or derived memory layouts using the best hand optimised kernels.

9.3 Production Runs of CENTORI

Though this project has focused on CENTORI as an HPC application CENTORI is primarily a tool for scientific research. The CCFE received a grant of HPCx processor time of 416 000 processor hours in 2009 through EPSRC grant EP/G041814/1. This was the first full scale test of the new modularised framework and the library implementation. The time was used to validate the model and investigate some of the features of the plasma. Figure 9.1 shows the results of a simulation of MAST after 1.525 ms of model time using a 0.5ns timestep on a $128 \times 64 \times 32$ grid using $8 \times 4 \times 4$ processors. It plots the poloidal ion flow velocity and exhibits some typical features of turbulence within the plasma.

Simulations using CENTORI continue into 2010 with further grants of time on HPC-FF and HECToR. The improved runtime performance of the code increases the value of these grants by allowing a faster evolution of the model and greater total run lengths.

9.4 Further Work

A possible extension to CENTORI is to create a hybrid OpenMP and MPI version that uses the shared memory resource available on a node rather than relying solely on MPI. This could be implemented by adding directives to the kernels in the library code and plugging the resulting module back into the main CENTORI. However, as the library code is not 100% of the serial runtime, it reduces the shared memory parallel proportion of the code and, as a consequence of Amdahl's law, may inhibit scaling versus the pure MPI version. To see the greatest advantages of this hybrid version the library should be extended to include as many of the other computational sections of the code as possible. Otherwise these would have to be parallelised independently, breaking the abstraction. This technique becomes more attractive as the number of processors per "node" becomes greater and the potential of congestion over the network increases.

The new modular framework allows CENTORI to be quickly adapted to emerging architectures and processor technologies. For example, it is believed there is a great potential for

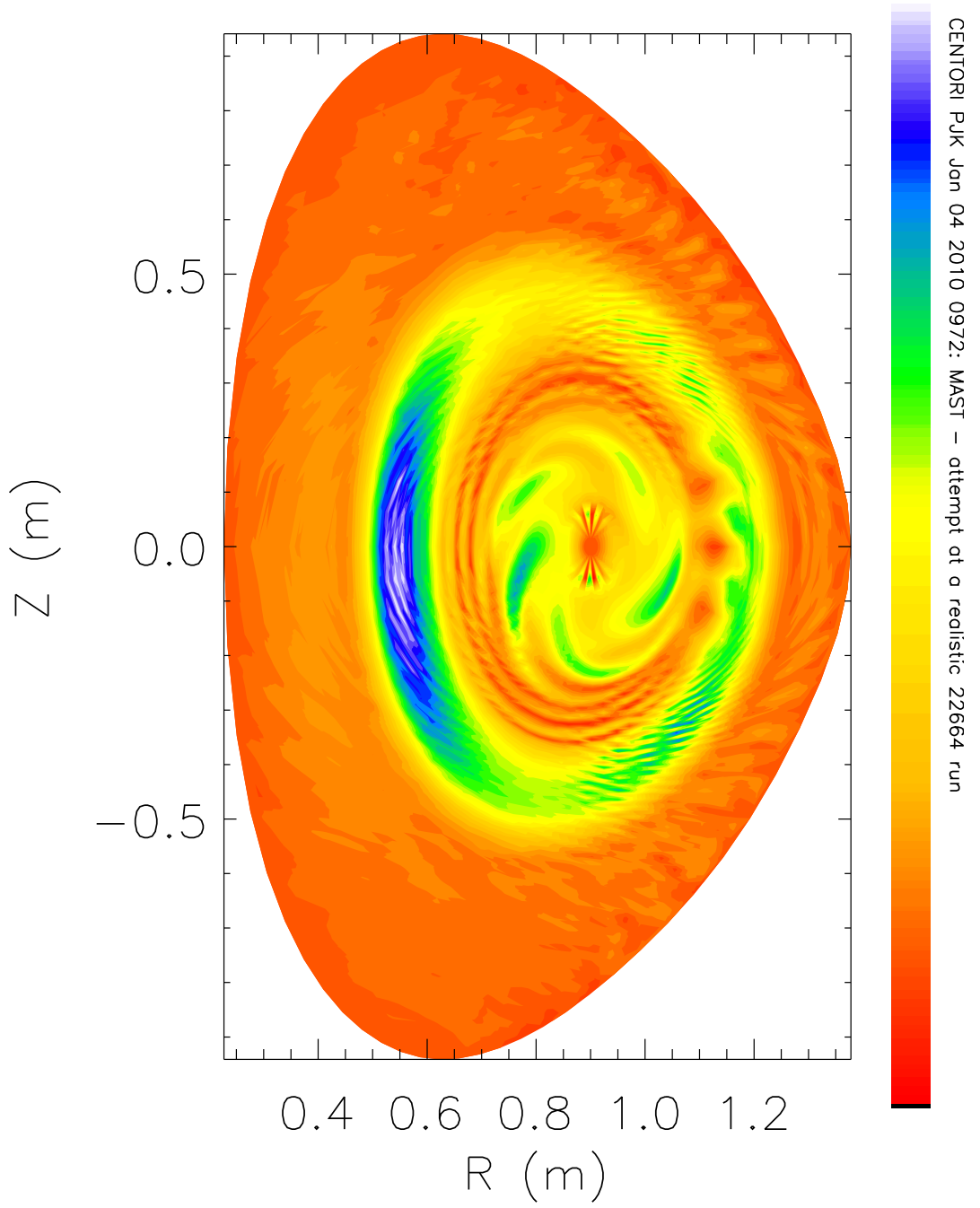


Figure 9.1: A snapshot of the poloidal ion flow velocity in a MAST plasma, calculated by CENTORI. Visible are features typically generated in the presence of turbulence within the plasma; the patterns correlate well with the topology of the local twist in the magnetic field. (Image and description courtesy of Peter Knight - CCFE)

Graphics Processing Units to be used as accelerators for floating-point kernels. A. Richardson has produced an early port of the benchmark kernels from Chapter 4 to an Nvidia's GPU processor as part of an MSc project at the University of Edinburgh[91]. This early work concluded that the kernels used in CENTORI can be adapted to run efficiently on GPUs but the cost of repeatedly transferring data between co-processor and the CPU erodes much of the advantage. Performing more calculations while the data is still on the co-processor card would improve this situation. A new implementation of the library could be used to port CENTORI for use on GPUs and the lazy evaluation scheme applied to reduce traffic between the card and the CPU. Though this may not, in the first instance, be an optimal port to GPUs, it should overcome some of the problems encountered by Richardson.

9.4.1 Automated Tuning

Developing a new library for each new architecture, even though the scope is much more limited thanks to the modular approach, still represents a significant amount of effort if done by hand. Projects like the Automatically Tuned Linear Algebra Software (ATLAS)[92] and the Fastest Fourier Transform in the West (FFTW)[93] libraries are capable of automatically tuning themselves for optimal performance on the target architecture.

Applying this approach to CENTORI would reduce the need for hand optimised libraries for each target architecture; instead the same self-tuning library could be used on every architecture. This could potentially reduce the time between CENTORI first running on a platform and achieving optimal performance to the time it takes to perform the tuning algorithm.

A simple approach to automatic tuning is to exhaustively search through a collection of implementations and parameters to find the combination that performs best on the target architecture. Early investigations suggested the Parameterized Optimizations for Empirical Tuning (POETS)[94] could be used to generate a large library of potential implementations. The project transforms fragments of source code automatically using methods that are equivalent to common hand optimisations. Time restrictions prevented investigating this area in more detail.

Tools and domain specific compilers that focus specifically on Linear Algebra routines, like Build-to-order[80] linear algebra or Digital Signal Processing like Spiral[81], may be able to generate optimised kernels for a new platform faster and with less effort than a human, so warrant further investigation.

9.5 Summary

In summary:

- The modularisation and optimisation work has allowed CENTORI's developers to be more productive; the code runs faster while maintaining its scaling and is easier to maintain or extend.
- Hand optimising the high level source code still offers performance improvements over using naive versions on all the architectures tested. Therefore the choice of kernels that implement the library will be an important factor in the resulting performance of CENTORI on each architecture.
- CENTORI can be adapted to run optimally on the resources available with only small changes to the configuration. Even with the choice of two implementations it is clear the eager scheme is most suitable for larger processor decompositions (i.e 1024 processors) while the lazy stripmining scheme best for smaller numbers of processors (i.e. 64).
- It is expected that CENTORI could be easily adapted to most changes in processor or architecture, like GPUs or other co-processors, by only making changes to the library implementation. This makes porting CENTORI a much simpler task in the future.

Bibliography

[1] L. Bernstein, R. Pachauri, and A. Reisinger, "Climate change 2007: Synthesis report," *Intergovernmental Panel on Climate Change*, p. 104, 2007. [Online]. Available: http://www.ipcc.ch/publications_and_data/publications_ipcc_fourth_assessment_report_synthesis_report.htm 1

[2] "The uk renewable energy strategy," Department of Energy and Climate Change, Tech. Rep., 2009. 1

[3] "European climate change policy beyond 2012," World Energy Council, Tech. Rep., 2009. 1

[4] J. Watson, C. Gough, and J. A. Hertin, "Renewable energy and combined heat and power resources in the uk," Tyndall Centre, Working Paper, 2002. 1

[5] D. J. MacKay, *Sustainable Energy - without the hot air*. UIT, 2008. 1

[6] J. Watson and A. Scott, "New nuclear power in the uk: A strategy for energy security?" *Energy Policy*, vol. 37, no. 12, pp. 5094 – 5104, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/B6V2W-4X0W4TG-2/2/034aa732f56c8d1de9fc580db7fe6cdc 1

[7] J. Fargione, J. Hill, D. Tilman, S. Polasky, and P. Hawthorne, "Land clearing and the biofuel carbon debt," *Science*, vol. 319, pp. 1235–1238, 2008. 1

[8] D. Eckhartt, "Nuclear fuels for low-beta fusion reactors: Lithium resources revisited," *Journal of Fusion Energy*, vol. 14, pp. 329–341, 1995. 1, 1.1.2

[9] I. Cook, G. Marbach, L. D. Pace, C. Girard, and N. P. Taylor, "Safety and environmental impact of fusion report," EFDA, Tech. Rep., 2001. 1

[10] J. Raeder, "Report on the european safety and environmental assesment of fusion power (seafp)," *Fusion Engineering and Design*, vol. 29, pp. 121–140, 1995. 1

[11] I. Cook, G. Marbach, L. D. Pace, C. Girard, P. Rocco, and N. P. Taylor, "Results, conclusions, and implications of the seafp-2 programme," *Fusion Engineering and Design*, vol. 51-52, pp. 409–417, 2000. 1

[12] J. D. Lawson, "Some Criteria for a Power Producing Thermonuclear Reactor," *Proceedings of the Physical Society B*, vol. 70, p. 6, 1956. 1.1.1

[13] J. Ongena and G. V. Oost, "Energy for future centuries will fusion be an inexhaustible, safe and clean energy source?" 1.1.2

[14] J. D. Lindl, P. Amendt, R. L. Berger, S. G. Glendinning, S. H. Glenzer, S. W. Haan, R. L. Kauffman, O. L, Landen, and L. J. Suter, "The physics basis for ignition using indirect-drive targets on the national ignition facility," *Physics of Plasmas*, vol. 11, pp. 340–479, 2004. 1.1.3

[15] E. I. Moses, "Ignition on the national ignition facility:a path towards inertial fusion energy," *Nuclear Fusion*, vol. 49, 2009. 1.1.3

[16] M. Keilhacker, A. Gibson, C. Gormezano, P. Lomas, P. Thomas, M. Watkins, P. Andrew, B. Balet, D. Borba, C. Challis, I. Coffey, G. Cottrell, H. D. Esch, N. Deliyanakis, A. Fasoli, C. Gowers, H. Guo, G. Huysmans, T. Jones, W. Kerner, R. Konig, M. Loughlin, A. Maas, F. Marcus, M. Nave, F. Rimini, G. Sadler, S. Sharapov, G. Sips, P. Smeulders, F. Soldner, A. Taroni, B. Tubbing, M. von Hellermann, D. Ward, and J. Team, "High fusion performance from deuterium-tritium plasmas in jet," *Nuclear Fusion*, vol. 39, no. 2, pp. 209–234, 1999. [Online]. Available: http://stacks.iop.org/0029-5515/39/209 1.1.5

[17] R. Aymar, V. Chuyanov, M. Huguet, Y. Shimomura, ITER Joint Central Team, and ITER Home Teams, "Overview of iter-feat - the future international burning plasma experiment," *Nuclear Fusion*, vol. 41, pp. 1301–1310, 2001. 1.1.5

[18] A. Sykes, J.-W. Ahn, R. Akers, E. Arends, P. G. Carolan, G. F. Counsell, S. J. Fielding, M. Gryaznevich, R. Martin, M. Price, C. Roach, V. Shevchenko, M. Tournianski, M. Valovic, M. J. Walsh, and H. R. W. M. Team, "First physics results from the mast mega-amp spherical tokamak," vol. 8, no. 5. AIP, 2001, pp. 2101–2106. [Online]. Available: http://link.aip.org/link/?PHP/8/2101/1 1.1.5

[19] D. Maisonnier, I. Cook, S. Pierre, B. Lorenzo, D. P. Luigi, G. Luciano, N. Prachai, and P. Aldo, "Demo and fusion power plant conceptual studies in europe," *Fusion Engineering and Design*, vol. 81, no. 8-14, pp. 1123 – 1130, 2006, proceedings of the Seventh International Symposium on Fusion Nuclear Technology - ISFNT-7 Part B. [Online]. Available: http://www.sciencedirect.com/science/article/B6V3C-4HVF0RX-X/2/a343c63930ad9fe943c8499cb2779582 1.1.5

[20] R. J. Hastie, *Plasma Physics An Introductory Course.* Cambridge University Press, 1993, ch. 1, pp. 5–28. 1.2.1

[21] W. M. Stacey, *Fusion Plasma Physics.* Wiley-Vch, 2005. 1.2.1, 2.7

[22] G. Falkovich and K. R. Sreenivasan, "Lessons from Hydrodynamic Turbulence," *Physics Today*, vol. April, pp. 43–49, 2006. 1.2.1

[23] K. Itoh, "Summary: theory of magnetic confinement," *Nuclear Fusion*, vol. 43, pp. 1710–1719, 2003. 1.2.1

[24] M. Kotchenrether, G. Rewoldt, and W. M. Tang, "Comparison of initial value and eigenvalue codes fro kinetic toroidal plasma instabilities," *Comp. Phys. Comm*, vol. 88, p. 128, 1995. 1.3.1

[25] [Online]. Available: http://gs2.sourceforge.net/ 1.3.1

[26] [Online]. Available: http://www.ipp.mpg.de/~fsj/gene/ 1.3.1

[27] M. R. de Baar, A. Thyagaraja, G. M. D. Hogeweij, P. J. Knight, and E. Min, "Global Plasma Turbulence Simulations of q = 3 Sawtoothlike Events in the RTP Tokamak," *Physical Review Letters*, vol. 94, 2005. 1.4

[28] M. P. I. Forum, *MPI: A Message-Passing Interface Standard.* Message Passing Interface Forum, 1997. 1.4

[29] P. J. Knight, "CENTORI Source Code Documentation." 1.4.1, 1.4.2, 1.2, 9.5

[30] R. Courant, K. Friedrichs, and H. Lewy, "On the partial difference equation of mathematical physics," *IBM Journal of Research and Development*, vol. 11, no. 2, p. 215, 1967. 1.4.4

[31] E. Minn, "Self-organisation in tokamak turbulence," Ph.D. dissertation, Technische Universiteit Eindhoven, 2006. 1.1, 9.5

[32] R. J. Buttery, O. Sauter, R. Akers, M. Gryaznevich, R. Martin, C. D. Warrick, H. R. Wilson, and the MAST Team, "Neoclassical tearing physics in the spherical tokamak mast," *Phys. Rev. Lett.*, vol. 88, no. 12, p. 125005, Mar 2002. 1.1, 9.5

[33] A. Thyagaraja, "Personal communicaiton: Discussion on energy loss using implicit and explicit methods," August 2009. 1.4.5

[34] R. L. Burden and J. D. Faires, *Numerical Analysis.* PWS-KENT, 1989. 1.4.5, 1.4.6

[35] W. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Receipes In Fortran.* Cambridge University Press, 1992. 1.4.5, 1.4.6

[36] T. Edwards, "Unpublished report: Performing numerical derviatives using the fast fourier transform," December 2009. 1.4.6

[37] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, no. 8, 1965. 2

[38] [Online]. Available: http://www.hector.ac.uk 2

[39] [Online]. Available: http://www.fz-juelich.de/jsc/juropa/configuration/ 2

[40] [Online]. Available: http://www.hpcx.ac.uk/ 2

[41] E. Strohmaier, J. J. Dongarra, H. W. Meuer, and H. D. Simon, "Recent trends in the marketplace of high performance computing," *Parallel Computing*, vol. 31, no. 3-4, pp. 261 – 273, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/B6V12-4G1GF38-1/2/cba5d2d0e6949c4d07ae13c8d380acc2 2.1

[42] *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual*, Intel. [Online]. Available: http://www.intel.com/design/intarch/manuals/243191.htm 2.1

[43] IBM, *Power ISA User Instruction Set Architecture*. IBM, 2007. 2.1

[44] AMD, "Quad-core and enhanced quad-core amd opteron processor general faq." [Online]. Available: http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_8796_8806~119722,00.html 2.1.2

[45] K. Aakre and G. Alfs, "Intel developer forum day 1 news disclosures," *News Fact Sheet*, September 2007. [Online]. Available: http://download.intel.com/pressroom/kits/events/idffall_2007/Day1FactSheet.pdf 2.1.2

[46] "Design team of boeing 747 wins third francois-xavier bagnoud aerospace prize," *Boeing News Release*, 1997. [Online]. Available: http://www.boeing.com/news/releases/1997/news.release.970521a.html 2.1.2

[47] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," *Computer Architecture, International Symposium on*, vol. 0, p. 0025, 2002. 2.2.1

[48] J. Stokes, *Inside the Machine: An Illustrated Introduction to Microprocessors and Computer Architecture*. No Starch Press, 2007. 2.2.1, 2.2.2, 2.2.3

[49] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach 4th Edition*. Elsevier, 2007. 2.2.1, 2.2.2, 2.2.3, 2.3.1

[50] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, 2008. 2.2.4

[51] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: maximizing on-chip parallelism," in *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1995, pp. 392–403. 2.2.4

[52] D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, pp. 56–65, 2003. 2.2.4

[53] M. Buxton, S. Kuo, K. Nasri, N. Firasta, and P. Jinbo, "Intel avx: New frontiers in performance improvments and energy efficiency," *Intel Software Network*, 2008. 2.2.5

[54] V. Babka and P. TÅ¯ma, "Investigating cache parameters of x86 family processors," *Computer Performance Evaluation and Benchmarking*, vol. 5419, pp. 77–96, 2009. [Online]. Available: http://www.springerlink.com/content/1180h7p14nuk72p8/ 2.3.1

[55] *Software Optimisation Guide for AMD Family 10h Processors*, 3rd ed., May 2009, publication 40546. 2.3, 3.4.1, 3.4.2, 9.5

[56] M. Hill and A. Smith, "Evaluating associativity in cpu caches," *IEEE Transactions on Computers*, vol. 38, pp. 1612–1630, 1989. 2.3.1

[57] J.-L. Baer and W.-H. Wang, "On the inclusion properties for multi-level cache hierarchies," *SIGARCH Comput. Archit. News*, vol. 16, no. 2, pp. 73–80, 1988. 2.3.1

[58] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache prefetch buffers," in *ISCA '98: 25 years of the international symposia on Computer architecture (selected papers)*. New York, NY, USA: ACM, 1998, pp. 388–397. 2.3.1, 2.3.2

[59] S. Pan, C. Cherng, K. Dick, and R. E. Ladner, "Algorithms to take advantage of hardware prefetching," in *the Workshop on Algorithm Engineering and Experiments (ALENEX*, 2007. 2.3.2

[60] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," *SIGPLAN Not.*, vol. 27, no. 9, pp. 62–73, 1992. 2.3.2

[61] D. Callahan, K. Kennedy, and A. Porterfield, "Software prefetching," *SIGARCH Comput. Archit. News*, vol. 19, no. 2, pp. 40–52, 1991. 2.3.2

[62] K. Dowd and C. Severance, *High Performance Computing*. O'Reilly, 1998. 2.4.2

[63] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@home: an experiment in public-resource computing," *Commun. ACM*, vol. 45, no. 11, pp. 56–61, 2002. 2.6

[64] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande, "Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology," 2009. [Online]. Available: http://www.citebase.org/abstract?id=oai:arXiv.org:0901.0866 2.6

[65] D. Stainforth, J. Kettleborough, A. Martin, A. Simpson, R. Gillis, A. Akkas, R. Gault, M. Collins, D. Gavaghan, and M. Allen, "Climate*prediction*.net: design principles for public resource modelling research." in *Proc. 14th IASTED conference on parallel and distributed computing systems.*, 2002. 2.6

[66] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: the architecture and performance of roadrunner," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 1–11. 2.6

[67] A. S. Bland, R. A. Kendall, D. B. Kothe, J. H. Rogers, and G. M. Shipman, "Jaguar: The world's most powerful computer," in *Proceedings Cray User Group 2009*, June 2009. 2.6

[68] M. J. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. C-21, no. 9, pp. 948 –960, Sept. 1972. 2.6.1

[69] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*. New York, NY, USA: ACM, 1967, pp. 483–485. 2.7.1

[70] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *Memory Bandwidth and Machine Balance in Current High Performance Computers*, December 1995. 3.1.1

[71] [Online]. Available: http://software.intel.com/en-us/articles/intel-mpi-benchmarks/ 3.1.2

[72] *Intel 64 and IA-32 Architectures Optimisation Reference Manual*, Intel, November 2009, order Number: 248966-020. 3.4.1

[73] O. Kaser, E. L, and C. R. Ramakrishnan, "Evaluating inlining techniques," *Computer Languages*, vol. 24, pp. 24–55, 1998. 4.1.1

[74] B. Calder, D. Grunwald, and B. Zorn, "Quantifying behavioral differences between C and C++ programs," *Journal of Programming Languages*, vol. 2, pp. 313–351, 1994. 4.1.1

[75] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff, "A set of level 3 basic linear algebra subprograms," *ACM Trans. Math. Softw.*, vol. 16, no. 1, pp. 1–17, 1990. 4.3

[76] O. Richards and M. Baker, "Gridpp and the edinburgh compute and data facility of how a general purpose cluster bore the weight of atlas on its shoulders," in *Proce. UK All Hands Meeting*, 2008. 4.4.1

[77] R. C. Whaley and A. M. Castaldo, "Achieving accurate and context-sensitive timing for code optimization," *Softw. Pract. Exper.*, vol. 38, no. 15, pp. 1621–1642, 2008. 4.4.4

[78] [Online]. Available: http://www.mathworks.com/products/matlab/ 5.1.2

[79] S. U. Aho Lam, *Compilers: Principles, Techniques, & Tools*. Addison Wesley, 2007. 5.3.6

[80] G. Belter, E. Jessup, I. Karlin, and J. Siek, "Automating the Generation of Composed Linear Algebra Kernels," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. Portland, Oregon: ACM Press, 2009, pp. 1–12. [Online]. Available: http://ecee.colorado.edu/~{}siek/sc09.pdf 6.3, 9.4.1

[81] M. Puschel, J. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. Johnson, and N. Rizzolo, "SPIRAL: Code Generation for DSP Transforms," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, February 2005. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1386651 6.3, 9.4.1

[82] M. Wolfe, "More iteration space tiling," in *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 1989, pp. 655–664. 6.3, 6.3.1

[83] K. Kennedy and K. S. McKinley, "Optimizing for parallelism and data locality," in *ICS '92: Proceedings of the 6th international conference on Supercomputing*. New York, NY, USA: ACM, 1992, pp. 323–334. 6.3.1

[84] D. B. Loveman, "Program improvement by source-to-source transformation," *J. ACM*, vol. 24, no. 1, pp. 121–145, 1977. 6.3.1

[85] M. Wolfe, "Loops skewing: The wavefront method revisited," *International Journal of Parallel Programming*, vol. 15, pp. 279–293, August 1986. 6.4.2

[86] F. Irigoin and R. Triolet, "Supernode partitioning," in *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, 1988, pp. 319–329. 6.4.2

[87] A. W. Burks, D. W. Warren, and J. B. Wright, "An analysis of a logical machine using parenthesis-free notation," *Mathematical Tables and Other Aids to Computation*, vol. 8, no. 46, pp. 53–57, 1954. [Online]. Available: http://www.jstor.org/stable/2001990 6.5.2

[88] J. Jones, "Abstract syntax tree implmentation idioms," in *10th Conference on Pattern Languages of Programs*, 2003. 6.5.2

[89] B. D. Garner, S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A portable programming interface for performance evaluation on modern processors," *The International Journal of High Performance Computing Applications*, vol. 14, pp. 189–204, 2000. 7.1

[90] R. G. Edwards and B. Joo, "The chroma software system for lattice qcd," *Nuclear Physics B - Proceedings Supplements*, vol. 140, pp. 832 – 834, 2005, lATTICE 2004 - Proceedings of the XXIInd International Symposium on Lattice Field Theory. [Online]. Available: http://www.sciencedirect.com/science/article/B6TVD-4F7YGGS-7M/2/02c8ac2edea772e09618f123d4f4abaf 9.2

[91] A. Richardson, "Gpu acceleration of hpc applications," Master's thesis, University of Edinburgh, August 2009. [Online]. Available: http://www.epcc.ed.ac.uk/wp-content/uploads/2009/10/Alan_Richardson.pdf 9.4

[92] R. C. Whaley and A. Petitet, "Minimizing development and maintenance costs in supporting persistently optimized BLAS," *Software: Practice and Experience*, vol. 35, no. 2, pp. 101–121, February 2005. [Online]. Available: http://www.cs.utsa.edu/~whaley/papers/spercw04.ps 9.4.1

[93] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation". 9.4.1

[94] Q. Yi, S. K, H. You, R. Vuduc, and D. Quinlan, "Poet: Parameterized optimization for empirical tuning," in *Workshop on Performance Optimization of High-Level Languages and Libraries*. IEEE Computer Society, March 2007. 9.4.1

List of Examples

List of Tables

List of Figures

Appendix A

The Interface

This section is an overview of the application programming interface and describes the core features and their intended use. Full details, including a specification for every function, are included here. The best validation of a new implementation of the interface is to compare the results against an appropriate version of CENTORI or against a reference implementation.

A.1 Preliminary Notes

Fortran Modules

Access to the subroutines and data types is through the Fortran module, `centori_fast_vectors`, which should be included by every program unit using fundamental data types of subroutines.

```
use centori_fast_vectors
```

Coordinate Systems

Vector objects are held as coefficients of one of three potential coordinate basis sets, the covariant, contravariant and physical basis sets: All functions dealing with vectors are capable of intelligently switching between representations if required, for example when performing a curl operation. This limits the number of circumstances where the internal representation of the vectors is actively specified.

Example A.1 Unassigning a locally scoped object.

```
subroutine process(input)

  implicit none

  type(sca_fld) :: input ! Input argument
  type(sca_pln) :: tmp  ! Local scope variable

  ! Some functionality

  call unassign(tmp)! tmp must be unassigned as it
                    ! is only local scope
end subroutine process
```

Basic Data types

There are four basic data types in CENTORI: the three dimensional vector field, three dimensional scalar field, two dimensional scalar field in the ψ and θ plane and the scalar profile in the ψ direction.

- `vec_fld` - a 3d $(n_\psi \times n_\theta \times n_\zeta)$ field of vector quantities. This field can be represented in coefficients of one of three basis sets, physical, covariant or contravariant.
- `sca_fld` - a 3d $(n_\psi \times n_\theta \times n_\zeta)$ field of a scalar quantity.
- `sca_pln` - a 2d field $(n_\psi \times n_\theta)$ field of a scalar quantity.
- `sca_prf` - a 1d list (n_ψ) of a scalar quantity.

These are derived data types in Fortran with no specification about what they should contain. They may be manipulated in the same way as intrinsic data types e.g. added, multiplied, assigned. Additionally, however if an object is created as local variable in a subroutine it must be unassigned when the procedure returns by calling

- `unassign(obj)` - where `obj` is any variable of type `vec_fld`, `sca_fld`, `sca_pln` or `sca_prf` which will no longer be referenced after the program unit returns.

Example A.1 shows its use. Failure to properly deallocate objects in this way will result in memory leaks.

Buffer Shapes and Sizes

This interface is designed to manipulate data on an individual MPI task and so represents only a subsection of the overall field when operating on more than one processor. The values n_ψ,

n_θ and n_ζ represent the number of points, including halos held on an individual processor. All buffers unless specified are arrays of double precision floating-point values. Buffers are always indexed in "array" form (as described in Chapter 4) and sized appropriately for the object, i.e $(n_\psi \times n_\theta \times n_\zeta \times 3)$ for a `vec_fld` object, $(n_\psi \times n_\theta \times n_\zeta)$ for a `sca_fld` object, $(n_\psi \times n_\theta)$ for a `sca_pln` object and (n_ψ) for a `sca_prf` object.

Buffers for purely halo data must also be appropriately sized, the size of the halo $\times$ the length of the remaining two axes, e.g. $halo_\psi \times n_\theta \times n_\zeta$ for the ψ halo and multiplied by 3 for `vec_fld` objects.

A.2 Global Constants

There are multiple flags defined as integer parameters which are required by the interface when passing various arguments. The values of these coordinates are not set by the interface and are free to change from implementation to implementation.

- `PHYSICAL`, `COVARIANT` and `CONTRAVARIANT` - Tokens representing the coordinate system for `vec_fld` objects.
- `COVARPSI`, `COVARTHETA` and `COVARZETA` - Tokens representing the component of a `vec_fld` object in `COVARIANT` form.
- `CONTRAPSI`, `CONTRATHETA` and `CONTRAZETA` - Tokens representing the component of a `vec_fld` in `CONTRAVARIANT` form.
- `RAD`, `POL` and `TOR` - Token representing the components of a `vec_fld` in `PHYSICAL` form.

A.3 Creating objects

An object can be initialised from an input array or to a single value. These functions will return a fully initialised object which can be used or assigned as necessary.

- `initialise_vec_fld(coords, [buffer|const])` - returns a vector field object fully initialised with the contents of `buffer` (see A.1) or `const` in representation specified by `coords`.
- `initialise_sca_[fld|pln|prf]([buffer|const])` - returns the appropriate object fully initialised with the contents of `buffer` (see A.1) or `const`.

All operands to functions should be initialised, either as the result of a previous operation or directly using the commands above. Passing uninitialised operands will result in undefined behaviour.

A.4 Input and Output Operations

The design of an API is a trade off between flexibility and size: it is possible to have a extensive API which can handle any case but is difficult to optimise, or a simple API that is fast but does not have the necessary functionality. This interface provides routines which are common in CENTORI and provides the following calls to allow data to be extracted from and inserted into objects for cases where additional flexibility is required. There is an inevitable performance penalty associated with this and so should be done only when necessary.

Whole object operations

- `get_vec_fld(field, buffer, coords)` - returns with `buffer` (see A.1) filled with the values of `field`, `coords` the coordinate system the `buffer` is in.
- `set_vec_fld(field, buffer, coords)` - fills `field` with values from `buffer` (see A.1) and sets coordinate system to `coords`. `field` should have been initialised previously.
- `get_[sca|vec]_[fld|prf|pln](obj, buffer)` - returns with `buffer` (see A.1) filled with the values of contents of `obj`.
- `set_[sca|vec]_[fld|prf|pln](obj, buffer)` - sets the value of `obj` to the values of `buffer` (see A.1). `obj` should have been initialised previously.

Single point operations

Each object can be manipulated on a single point basis by calling subroutines which can get and set values.

- `get_point_vector(field, position, buffer, coords)` - returns with the array `buffer` filled with the values of the point referenced by `position` (ψ, θ, ζ) indexed from 0. The `buffer` is filled in order (`COVARPSI`, `COVARTHETA`, `COVARZETA`) if `coords` is set to `COVARIANT`, (`CONTRAPSI`, `CONTRATHETA`, `CONTRAZETA`) if `coords` equals `CONTRAVARIANT` and (`RAD`, `POL`, `TOR`) if `coords` is `PHYSICAL`.

– `set_point_vector(field, position, buffer, coords)` - as above where the values of buffer are placed in the `vec_fld`, `field`, at the point referenced by `position` (indexed from 0).

– `set_point_sca_fld(field, position, buffer)` - sets the point in `field` referenced by array `position` (ψ, θ, ζ) to the value in `buffer`.

– `get_point_sca_fld(field, position)` - returns the value of the point indexed by the array `position` (ψ, θ, ζ) indexed from 0 in `field`.

– `[get|set]_point_pln(plane, position, buffer)` - gets/sets the point in the `plane` referenced by the 2d array `position` (ψ, θ) and either returns or sets the value from/in `buffer`.

– `[get|set]_point_prf(profile, position, [buffer])` - gets/sets the point in the `profile` reference by the integer `position` (ψ) either returns the value or setting to the value in `buffer`.

Halo operations

– `get_vector_[psi|theta|zeta]_halo_[up|dn](field, buffer, halo_depth, coords)` - returns the data from `field` that forms the upper or lower halo into `buffer`. This buffer must be correctly sized to `halo_depth` $\times$ (product of two remaining dimensions $\times 3$). The internal structure of buffer is left undefined except that it should be passed to the equivalent `set_vector_[psi|theta|zeta]_halo_[up|dn]` call on the neighbouring processor. The value of `coords` is set to the coordinate system the buffer's data is stored in.

– `set_vector_[psi|theta|zeta]_halo_[up|dn](field, buffer, halo_depth, coords)` - uses the data in `buffer` to update the halo regions of `field` in the appropriate object. Data in `buffer` must be from an equivalent call to `get_vector_[psi|theta|zeta]_halo_[up|dn]` on a homogeneous system. The buffer must be appropriately sized as above. Value of `coords` records the representation of the data held in buffer.

– `[get|set]_scalar_[psi|theta|zeta]_halo_[up|dn](field, buffer, halo_depth)` - equivalent to the vector halo operations for `sca_fld`.

Performance Considerations

The developer is at liberty to use any and all of these functions as appropriate and it is expected there is a balance to be struck between the efficiency of extracting data in a separate buffer and performing complicated or sophisticated manipulation which are difficult or impossible using the interface and performing them point by point. It is generally recommended that such copying is avoided wherever possible.

A.5 Manipulating Vector Representations

Each vector field can be represented as coefficients of three distinct sets of basis vectors: the contravariant, the covariant, and the physical. The library is intelligent enough that the developer does not need to actively keep track of the state of `vec_fld` as on-the-fly conversions are possible. It may, however, be necessary to convert a field manually (usually before `call get_vec_fld`). The operators to do this are:

- `to_physical(field)` - converts field to physical
- `to_contra(field)` - convert field to contravariant
- `to_covariant(field)` - convert field to covariant
- `to_coords(field, coords)` - convert field to parameter specified by coords
- `get_coords(field)` - return the current coordinate type of field

A.5.1 Type Transitions

Each of the objects vary in size considerably, with the `vec_fld` the largest and a `sca_prf` the smallest. When manipulating these objects it is often necessary to "upgrade" one type to a larger type and the interface provides methods for doing this.

Default Promotion Behaviour

When an object is promoted from one type to the next type up, by default the values of the small object are copied repeatedly across the extra dimension. For example a `sca_prf` promoted to a `sca_pln` will have the value for all theta coordinates for a particular ψ coordinate which in most cases is the desired behaviour (in CENTORI). When an object is promoted above the

next level up, multiple promotions occur using the default behaviour. It is possible for any object to be promoted to any other object up to including `vec_fld` using:

- `to_vec(object, coords)`
- `to_[sca|pln](object)`

There is no `to_prf` routine as the equivalent behaviour is to initialise a `sca_prf` with a constant. It is undefined behaviour to attempt to promote an object of higher or equivalent type to a lower type, e.g. `to_sca(vec_fld)`.

Alternative Promotion Behaviour

In some cases the repeating behaviour is not desired; instead an extension to the `to_*` operations is provided which allows objects to be promoted with only a single coordinate in the extra dimension receiving the values and the remainder being set to zero. For example, when promoting a `sca_pln` to a `sca_fld` it may be necessary for only the final zeta coordinate to have the values from the `sca_pln`.

- `to_vec(profile, coords, component, zeta, theta)` - When promoting from a `sca_prf`, `component`, `theta` and `zeta` are all optional.
- `to_vec(plane, coords, component, zeta)` - When promoting from a `sca_pln`, `zeta` and `component` are optional.
- `to_vec(field, coords, component)` - When promoting from a `sca_fld`, `component` is optional.
- `to_sca(profile, zeta, theta)` - When promoting from a `sca_prf`, `zeta` and `theta` are optional.
- `to_sca(plane, zeta)` - When promoting from a `sca_pln`, `zeta` is optional.
- `to_pln(object, theta)` - When promoting from a `sca_prf`, `theta` is optional.

As in the default case, attempting to promote an object to the same or lower type is undefined.

Cutting and Pasting objects

It may also be required to extract a lower type from a larger type, i.e. extracting a component of a `vec_fld` to a `sca_fld`. Operators are provided for this:

- `get_component(vec_fld, coords, component)` - returns a `sca_fld` which holds the values of the `vec_fld`'s component in the specified coordinate system.
- `set_component(vec_fld, sca_fld, coords, component)` - sets the `component` of `vec_fld` to the values held in `sca_fld` in the specified coordinate system.
- `get_sca_pln(field, zeta)` - extracts a `sca_pln` from the `zeta` component of `field`.
- `set_sca_pln(field, plane, zeta)` - inserts `plane` at the `zeta` component of `field`.
- `get_sca_prf(plane, theta)` - extracts a `sca_prf` from the `theta` component of `plane`.
- `set_sca_prf(plane, profile, theta)` - inserts `plane` at the `theta` component of `plane`.

A.6 Basic Mathematical Operations

Scalar Operations

The API specifies functions for performing basic mathematical operations with all operand type combinations by enforcing transparent promotion of objects when necessary using the default method defined in Section A.5.1. Conversions are always promotions so the result will always be an object of the higher type (see Table A.1). If an operation requires an object to be promoted using the alternative promotion method (see Section A.5.1) an explicit promotion call like `to_vec`, `to_sca` or `to_pln` is necessary. When one of the operands is a `vec_fld` and the other a scalar object the operation is performed in the coordinate system of the `vec_fld` object. If an operation must occur in a particular coordinate system an explicit conversion call is required (see A.5).

The basic binary operations are

- addition
- subtraction
- multiplication[1]

and the unary operators are

- reciprocal over a constant

[1]Multiplication of two vectors is provided (each component is multiplied) though it is not standard mathematical operation, traditional multiplicative operators cross and dot are defined in Section A.6.

	vec_fld	sca_fld	sca_pln	sca_prf	cnst
vec_fld	vec_fld	vec_fld	vec_fld	vec_fld	vec_fld
sca_fld	vec_fld	sca_fld	sca_fld	sca_fld	sca_fld
sca_pln	vec_fld	sca_fld	sca_pln	sca_pln	sca_pln
sca_prf	vec_fld	sca_fld	sca_pln	sca_prf	sca_prf
cnst	vec_fld	sca_fld	sca_pln	sca_prf	cnst

Table A.1: Object interaction.

Operation		In code
scalar product	$c = \mathbf{a}.\mathbf{b}$	`c = a .dot. b`
vector product	$\mathbf{c} = \mathbf{a} \times \mathbf{b}$	`c = a .cross. b`
L2 norm	$c = \lvert\mathbf{a}\rvert^2$	`c = l2(a)`

Table A.2: Vector Operators.

– square root

– modulus

The intrinsic operators are overloaded in Fortran so calculations can be expressed using standard `+`, `*`, `/`,`-` notation and the `abs` and `sqrt` function calls.

Vector Operators

Standard vector operations provided include the vector product, scalar product, L2 norm. These are mapped to operators as show in Table A.2

Stencil Operations

CENTORI requires unary stencil like operators which can difference or sum neighbours of a point. The interface provides functions which do this.

– `add_stncl_[sca|vec](field, direction, up, down)` - Sums the points `up` and `down` away from each point in the `direction` of `field`.

– `sub_stncl_[sca|vec](field, direction, up, down)` - Subtracts the points `up` and `down` away from each point in the `direction` of `field`.

These operations will produce undefined results for regions which are impossible to calculate by definition, points where there is no `up` or `down` point e.g. for a `sca_fld`, `in`, adding the nearest neighbours in the ψ direction the call would be `sum = add_stncl_sca(in, PSI_DIR, 1, 1)` and the result valid for the points $1 \leq \psi \leq npsi - 1$.

Example A.2 Three dimensional nearest neighbour sums.

```
sum = add_stncl_sca(in, PSI_DIR, 1, 1) + add_stncl_sca(in,
    THETA_DIR, 1, 1) + add_stncl_sca(in, ZETA_DIR, 1 1)
```

It is possible to perform more complicated stencil operations by summing multiple stencil operations, as shown in Example A.2

Any stencil operations on `vec_fld`s will be calculated in physical coordinates, as it is the only consistent basis set between points on the lattice.

A.7 Calculus Operations

CENTORI requires derivative operations on the native fields. The interface provides the following calls:

Scalar

Simple partial derivative operations are available along each of the three directions ψ, θ and ζ.

- `d_[psi|theta|zeta]_sca(field, dx)` - returns a `sca_fld` with the derivative of `field` in the appropriate direction assuming $\delta =$ `dx`.
- `d_[psi|theta]_pln(plane, dx)` - returns a `sca_pln` with the derivative of `plane` in the appropriate direction assuming $\delta =$ `dx`.
- `d_psi_prf(profile, dx)` - returns a `sca_prf` with the derivative of `profile` in the ψ direction assuming δ, `dx`.

Vector

Standard vector calculus operations are available: the curl, gradient and divergence operations on `vec_fld`. These operations require two additional tokens which specify how the axis and edge boundary conditions are to be filled. Each operation will perform a halo exchange after each call.

The reference implementation uses a 2nd order finite difference scheme which can be operated from the halo, however this is not required and other algorithms and methods could be applied.

Description	*Equation*	*Code*	*Input*	*Output*
curl	$\mathbf{b} = \nabla \times \mathbf{a}$	`b = curl(a, BC_AXIS, BC_EDGE)`	covariant	contravariant
gradient	$\mathbf{b} = \nabla a$	`b = grad(a, BC_AXIS, BC_EDGE)`	scalar	covariant
divergence	$b = \nabla.\mathbf{a}$	`b = div(a, BC_AXIS, BC_EDGE)`	contravariant	scalar

Table A.3: Vector calculus operators.

A note on forcing halo exchanges

The operations `curl`, `grad` and `div` force data to be exchanged between processors, because they destroy the halo data. Though this is potentially the case in the stencil operation cases and the scalar derivative operations, because the halo is only destroyed in one direction (either ψ, θ or ζ) for efficiency purposes it is left to the programmer to force a halo exchange when necessary.

Appendix B

Example Kernel Source Code

This appendix contains examples of the "naive" or "canonical" kernels using the "derived" memory layout as implemented as part of the benchmarks in Chapter 4

Listing B.1: Add Kernel

```
subroutine add(out, a, b)
   implicit none
   ! Input arguments
   type(vector), dimension(nx,ny,nz), intent(in)  :: a, b ! input vector fields
   type(vector), dimension(nx,ny,nz), intent(out) :: out ! input vector fields
   ! Local variables
   integer :: i, j, k
   do k=1,nz
     do j=1,ny
       do i=1,nx
         out(i,j,k)%x = a(i,j,k)%x + b(i,j,k)%x
         out(i,j,k)%y = a(i,j,k)%y + b(i,j,k)%y
         out(i,j,k)%z = a(i,j,k)%z + b(i,j,k)%z
       end do
     end do
   end do
 end subroutine
```

Listing B.2: Grad Kernel

```
subroutine grad(a, out)

   implicit none

   ! Input arguments
   real(kind(1.0D0)), intent(in) :: a(nx,ny,nz) ! input vector field

   ! Output arguments
   type(vector), dimension(nx,ny,nz), intent(out) :: out

   ! Local variables
   integer :: i, j, k

   do k=2,nz-1
     do j=2,ny-1
       do i=2,nx-1
         out(i,j,k)%x = (a(i+1,j,k)  - a(i-1,j,k)) * odx
         out(i,j,k)%y = (a(i,j+1,k)  - a(i,j-1,k)) * ody
         out(i,j,k)%z = (a(i,j,k+1)  - a(i,j,k-1)) * odz
       end do
     end do
   end do
 end subroutine
```

Listing B.3: Divergence Kernel

```
 subroutine div(a, jac, invjac, out)

   implicit none

   type(vector), dimension(nx,ny,nz), intent(in) :: a ! input vector field
   real(kind(1.0D0)), dimension(nx,ny), intent(in) :: jac, invjac

   ! Output arguments
   real(kind(1.0D0)), dimension(nx,ny,nz), intent(out) :: out(nx,ny,nz)

   integer :: i, j, k

   do k=2,nz-1
     do j=2,ny-1
       do i=2,nx-1
         out(i,j,k) = jac(i,j) * ((invjac(i+1,j)*a(i+1,j,k)%x  - invjac(i-1,j)*a(i-1,j,k)%x
              ) * odx + (invjac(i,j+1)*a(i,j+1,k)%y  - invjac(i,j-1)*a(i,j-1,k)%y) * ody +
              (invjac(i,j)*(a(i,j,k+1)%z  - a(i,j,k-1)%z)) * odz)
       end do
     end do
   end do
 end subroutine
```

Listing B.4: Curl Kernel

```
subroutine curl(a, jac, out)

  implicit none

  type(vector), dimension(nx,ny,nz), intent(in) :: a ! input vector field
  real(kind(1.0D0)), dimension(nx,ny), intent(in) :: jac

  ! Output arguments
  type(vector), dimension(nx,ny,nz), intent(out) :: out

  integer :: i, j, k

  do k=2,nz-1
    do j=2,ny-1
      do i=2,nx-1
        out(i,j,k)%x = jac(i,j) * ((a(i,j,k+1)%y  - a(i,j,k-1)%y) * odz - (a(i,j+1,k)%z -
            a(i,j-1,k)%z) * ody)
        out(i,j,k)%y = jac(i,j) * ((a(i+1,j,k)%z  - a(i-1,j,k)%z) * odx - (a(i,j,k+1)%x -
            a(i,j,k-1)%x) * odz)
        out(i,j,k)%z = jac(i,j) * ((a(i,j+1,k)%x  - a(i,j-1,k)%x) * ody - (a(i+1,j,k)%y -
            a(i-1,j,k)%y) * odx)
      end do
    end do
  end do
end subroutine
```

Listing B.5: Convert Kernel

```
subroutine convert(a,metric,out)

  implicit none

  type(vector), dimension(nx,ny,nz), intent(in) :: a, metric
  type(vector), dimension(nx,ny,nz), intent(out) :: out

  integer :: i,j,k

  do k=1,nz
    do j=1,ny
      do i=1,nx
        out(i,j,k)%x = a(i,j,k)%x * metric(i,j,CYC_X)%x + a(i,j,k)%y * metric(i,j,CYC_X)%y
            + a(i,j,k)%z * metric(i,j,CYC_X)%z
        out(i,j,k)%y = a(i,j,k)%x * metric(i,j,CYC_Y)%x + a(i,j,k)%y * metric(i,j,CYC_Y)%y
            + a(i,j,k)%z * metric(i,j,CYC_Y)%z
        out(i,j,k)%z = a(i,j,k)%x * metric(i,j,CYC_Z)%x + a(i,j,k)%y * metric(i,j,CYC_Z)%y
            + a(i,j,k)%z * metric(i,j,CYC_Z)%z
      end do
    end do
  end do
end subroutine
```

Listing B.6: Dot Kernel

```
subroutine dot(a,b,out)

  implicit none

  type(vector), dimension(nx,ny,nz), intent(in) :: a,b
  real(kind(1.0D0)), dimension(nx,ny,nz), intent(out) :: out
  integer :: i,j,k

  do k=1,nz
    do j=1,ny
      do i=1,nx
        out(i,j,k) = (a(i,j,k)%x * b(i,j,k)%x) + (a(i,j,k)%y * b(i,j,k)%y) + (a(i,j,k)%z *
            b(i,j,k)%z)
      end do
    end do
  end do
end subroutine
```

Listing B.7: L2 Norm Kernel

```
subroutine l2(a,out)

  implicit none

  type(vector), dimension(nx,ny,nz), intent(in) :: a
  real(kind(1.0D0)), dimension(nx,ny,nz), intent(out) :: out
  integer :: i,j,k

  do k=1,nz
    do j=1,ny
      do i=1,nx
        out(i,j,k) = (a(i,j,k)%x * a(i,j,k)%x) + (a(i,j,k)%y * a(i,j,k)%y) + (a(i,j,k)%z *
            a(i,j,k)%z)
      end do
    end do
  end do
end subroutine
```

Listing B.8: Cross Kernel

```
subroutine cross(a,b,jac,out)

  implicit none

  type(vector), dimension(nx,ny,nz), intent(in) :: a,b
  type(vector), dimension(nx,ny,nz), intent(out) :: out
  real(kind(1.0D0)), dimension(nx,ny), intent(in) :: jac
  integer :: i,j,k

  do k=1,nz
    do j=1,ny
      do i=1,nx
        out(i,j,k)%x = jac(i,j) * (a(i,j,k)%y * b(i,j,k)%z - a(i,j,k)%z * b(i,j,k)%y)
        out(i,j,k)%y = jac(i,j) * (a(i,j,k)%z * b(i,j,k)%x - a(i,j,k)%x * b(i,j,k)%z)
        out(i,j,k)%z = jac(i,j) * (a(i,j,k)%x * b(i,j,k)%y - a(i,j,k)%y * b(i,j,k)%x)
      end do
    end do
  end do
end subroutine
```

Listing B.9: Scaleadd Kernel

```
subroutine scaleadd(out, in1, factor1, in2, factor2)

  implicit none

  type(vector), dimension(nx,ny,nz), intent(in) :: in1, in2
  type(vector), dimension(nx,ny,nz), intent(out) :: out
  real(kind(1.0D0)), intent(in) :: factor1, factor2

  integer :: i,j,k

  do k=1,nz
    do j=1,ny
      do i=1,nx
        out(i,j,k)%x = (factor1 * in1(i,j,k)%x) + (factor2 * in2(i,j,k)%x)
        out(i,j,k)%y = (factor1 * in1(i,j,k)%y) + (factor2 * in2(i,j,k)%y)
        out(i,j,k)%z = (factor1 * in1(i,j,k)%z) + (factor2 * in2(i,j,k)%z)
      end do
    end do
  end do
end subroutine
```

Listing B.10: Scale Kernel

```
subroutine scale(a, b)

  implicit none

  ! Input arguments
  type(vector), dimension(nx,ny,nz), intent(inout) :: a ! input vector field
  real(kind(1.0D0)), intent(in) :: b

  integer :: i, j, k

  do k=1,nz
    do j=1,ny
      do i=1,nx
        a(i,j,k)%x = a(i,j,k)%x * b
        a(i,j,k)%y = a(i,j,k)%y * b
        a(i,j,k)%z = a(i,j,k)%z * b
      end do
    end do
  end do
end subroutine
```