High-level Abstractions for Performance, Portability and Continuity of Scientific Software on Future Computing Systems

G.R. Mudalige, I.Z. Reguly, M.B. Giles, W. Gaudin, A. Mallinson and O. Perks
Oxford e-Research Centre, University of Oxford,
Dept. of Computer Science University of Warwick
AWE, Reading UK.

April 2014

Abstract

In this report we present research on applying a domain specific high-level abstractions development strategy with the aim to “future-proof” a key class of high performance computing (HPC) applications that simulate hydro-dynamics computations at AWE plc. We build on an existing high-level abstraction framework, OPS, that is being developed for the solution of multi-block structured mesh-based applications at the University of Oxford. The target application, is an unclassified benchmark application, CloverLeaf, that consists of algorithms of interest from the hydro-dynamics workload at AWE plc.

OPS uses an “active library” approach where a single application code written using the OPS API can be transformed into different parallel implementations which can then be linked against the appropriate parallel library enabling execution on different back-end hardware platforms. At the same time the generated code and the platform specific back-end libraries are highly optimized utilizing the best low-level features of a target architecture to make an OPS application achieve near-optimal performance including high computational efficiency and minimized memory traffic.

We present (1) the conversion of CloverLeaf to utilize OPS, (2) the utilization of OPS’s code generation tools for obtaining several parallel implementations of the application and finally (3) its performance compared to the original application on a range of parallel systems including multi-core CPUs, NVIDIA GPUs and a Cray XC30 distributed memory system.

Our results show that the development of parallel HPC applications through the careful factorization of a parallel program’s functionality and implementation, through a high-level framework such as OPS is no more time consuming nor difficult than writing a one-off parallel program targeting only a single parallel implementation. However the OPS strategy pays off with a highly maintainable single application source without compromising performance portability on the parallel systems on which it will be executed. It also lays the groundwork for providing support for execution on future parallel systems. We believe such an approach will be an essential paradigm shift for utilizing the ever-increasing complexity of novel hardware/software technologies.
# Contents

1 **Introduction** 3

2 **OPS** 3
   2.1 The OPS API ........................................... 4
   2.2 Porting CloverLeaf to OPS ............................. 7

3 **Code Generation with OPS** 7
   3.1 Optimized Single-threaded CPU Version with Vectorization ............. 8
   3.2 OpenMP Parallelization for Multi-core CPUs .......................... 12
   3.3 CUDA Implementation for NVIDIA GPUs .......................... 15
   3.4 Distributed-memory Parallelization with MPI ....................... 18

4 **Performance** 20

5 **Conclusions and Future Work** 22
1 Introduction

For many years, increasing the clock frequency of microprocessors led to steady improvements in the performance of computer applications. This gave an almost free performance boost to the speed and throughput of applications without having to re-write software for each new generation of processors. However, having approached the physical limits of current CMOS based microprocessor technology, higher clock frequencies of processors were leading to significant increases in a processor’s energy consumption. As a result the above method of improving performance with higher clock rates became unsustainable.

To gain higher performance, chip developers now rely on multiple processors to operate in parallel. This strategy has led to the development of multi-core and many-core processors, where a single silicon chip contains multiple processor cores, both on mainstream CPUs and as add-on accelerators. As the parallel processor landscape changes rapidly, such emerging many-core and heterogeneous processor combinations pose major challenges for software development, particularly for production level high performance applications which takes years to develop, validate and optimize. Although application developers would like to benefit from the performance gains promised by these new processor systems, they are very worried about the software development costs involved. They cannot afford to continuously re-engineer the applications for each new architecture to maintain performance.

A solution to the above issue is to utilize a high-level abstractions approach in developing parallel applications. This allows the de-coupling of the scientific specification of the application from its parallel implementation enabling scientists and engineers to develop applications based on a domain specific API or language (DSL) without being concerned about its implementation. At the same time, a lower implementation level will provide opportunity for parallel programming experts to apply radically aggressive and platform specific optimizations when implementing the required solution on various processor systems. This approach will enable easy maintenance of the higher-level software source with near optimal performance for various platforms and makes it possible to easily integrate support for any future novel hardware.

In this document we present research on applying such a domain specific high-level abstractions strategy with the aim to “future-proof” a key class of high performance computing (HPC) applications used to simulate hydro-dynamics computations at AWE plc. We build on an existing high-level abstraction layer and framework, OPS [1] that is being developed for the solution of multi-block structured mesh-based applications. The target application, is an unclassified benchmark application (CloverLeaf [2]), that consists of algorithms of interest from the hydro-dynamics workload at AWE plc. Utilizing unclassified applications as proxies allows AWE to understand various scientific simulation challenges and their pertinence to real day-to-day problems.

More specifically we report on the following:

- Present an overview of the development of the OPS high-level framework, its API and code generation techniques.
- Detail the process of converting CloverLeaf to the OPS API and the subsequent code generation through OPS to obtain parallelized code. We generate code targeting OpenMP thread level multi-core parallelism, single-instruction multiple-thread (SIMT) many-core parallelism using NVIDIA CUDA and distributed memory parallelism with MPI.
- Benchmark the performance of the resulting parallel code on a range of single node CPU and GPU systems and a distributed memory Cray XC30. Benchmark systems include compute nodes consisting of the latest Intel Sandy Bridge and Ivy Bridge processors and NVIDIA Kepler generation GPUs (K20 and K40). The resulting performance will be contrasted with the original hand-parallelized (non-OPS) versions of the application.

The rest of this report is organized as follows: in Section 2 we present the OPS abstraction, its API and its design; in Section 3 we discuss in more detail the process of code generation and the optimisations carried out for each parallelization; in Section 4, a performance analysis and benchmarking study of the application is carried out comparing the OPS based CloverLeaf with the original hand-tuned version. Finally Section 5 notes future work and concludes the report.

2 OPS

Previous work carried out at the University of Oxford, in developing the OP2 [3] high-level abstraction framework targeting the domain of unstructured mesh based applications, demonstrated that both developer productivity as well as near-optimal performance could be achieved on a wide range of parallel
hardware through such high-level frameworks. Research published as a result of this work include a number of performance analysis studies on standard CFD benchmark applications [4] as well as a full industrial-scale application from the production work-load at Rolls-Royce plc. [5].

OP2 uses an “active library” approach where a single application code written using the OP2 API can be transformed in to different parallel implementations which can then be linked against the appropriate parallel library enabling execution on different back-end hardware platforms. At the same time the generated code from OP2 and the platform specific back-end libraries are highly optimized utilizing the best low-level features of a target architecture to make an OP2 application achieve near-optimal performance including high computational efficiency and minimized memory traffic.

With the success of OP2, our aim is to apply the same strategy in developing high-level frameworks for another key class of applications, multi-block structured mesh applications, that require significant developer effort to parallelize on modern hardware systems. The resulting research and development is being carried out under the EPSRC [6] funded OPS project (Oxford Parallel library for Structured-mesh solvers). Multi-block structured mesh applications can be viewed as an unstructured collection of structured mesh blocks. Thus the starting point of our work was to adopt the high-level framework and code generation strategy from OP2 and apply it for the development and solution of single block structured mesh applications.

OPS is designed to appear as a classical software library with an API. It is currently domain specific to the single block-structured mesh problems and will later be extended to multi-block structured problems. Much of the API and library follows the design of the OP2 high-level library for unstructured mesh applications [3]. However the structured mesh domain is distinct from the unstructured mesh applications domain due to the implicit connectivity between neighboring mesh elements (such as vertices, cells) in structured meshes/grids. The key idea is that operations involve looping over a “rectangular” multi-dimensional set of grid points using one or more “stencils” to access data. The next section illustrates the OPS API using examples from CloverLeaf.

2.1 The OPS API

The CloverLeaf mini-app involves the solution of the compressible Euler equations, which form a system of three partial differential equations. The equations are statements of the conservation of energy, density and momentum and are solved using a finite volume method on a structured staggered grid. The cell centers hold internal energy and density while nodes hold velocities. The solution involves an explicit Lagrangian step using a predictor/corrector method to update the hydrodynamics, followed by an advective remap step returns the grid to its original position. The original application [2] is written in Fortran and operates on a 2D structured mesh. It is of fixed size in both x and y dimensions.

OPS separates the specification of such a problem into four distinct parts: (1) structured blocks, (2) data defined on blocks, (3) stencils defining how data is accessed and (4) operations over blocks. Thus the first aspect of declaring such a single-block structured mesh application with OPS is to define the size of the regular mesh over which the computations will be carried out. In OPS vernacular this is called an ops_block. OPS declares a block with the ops_decl_block API call by indicating the dimension of the block (2D in this case), sizes in each dimension and assigning it a name for identification and runtime checks (see Figure 2).

CloverLeaf works on a number of data arrays (or fields) which are defined on the 2D structured mesh (e.g. density, energy, x and y velocity of the fluid). OPS allows users to declare these using the
```c
int dims[2] = {x_cells+5, y_cells+5};
/* Declare a single structured block */
ops_block cl_grid = ops_decl_block(2, dims, "clover grid");
int d_p[2] = {-2,-2};
int d_m[2] = {-2,-2};
int size[2] = {x_cells+5, y_cells+5};
double* dat = NULL;
/* Declare data on block */
double density0 = ops_decl_dat(cl_grid,1,size,d_m,d_p,dat,"double","density0");
double energy0 = ops_decl_dat(cl_grid,1,size,d_m,d_p,dat,"double","energy0");
double pressure = ops_decl_dat(cl_grid,1,size,d_m,d_p,dat,"double","pressure");
double volume = ops_decl_dat(cl_grid,1,size,d_m,d_p,dat,"double","volume");
```

**Figure 2:** OPS API example for declaring blocks, data and stencils

The `ops_decl_dat` API call; the `density0, energy0, ... pressure` and `vol` are `ops_dat`s that are declared through the `ops_decl_dat` API call. A key idea is that once a field’s data is declared via `ops_decl_dat` the ownership of the data is transferred to OPS from the user, where it is free to rearrange the memory layout as is optimal for the final parallelization and execution hardware.

In this example a NULL pointer of type `double` is passed as an argument. CloverLeaf initializes these values later, as part of the application itself. When a NULL array is supplied, OPS will internally allocate the required amount of memory based on the type of the data array and its size, including additional depths (`d_m` and `d_p`) for each dimension. On the other hand an array containing the relevant initial data can be used in declaring an `ops_dat`. In the future we will provide the ability to read in data from HDF5 files directly using a `ops_decl_dat_hdf5` API call. Note above in an `ops_decl_dat` call, a single double precision value per grid element is declared. A vector of a number of values per grid element could also be declared (e.g. a vector with three doubles per grid point to store x, y, and z velocities). The depths `d_m` and `d_p` are a current workaround that is used to indicate the offset from the edge (in both the negative and positive directions of each dimension). In the current API, it is important that the application developer indicate to OPS the maximum stencil depth a given `ops_dat` is accessed within any of the subsequent loops over a block. This is to account for stencils accessing beyond the defined regions of the `ops_dat` during an execution, particularly for distributed memory halo allocation within OPS. In the future OPS will be able to dynamically allocate the maximum necessary depths based on the stencils declared to be used on a given `ops_dat`, eliminating the need to declare `d_m` and `d_p`.

All the numerically intensive computations in the structured mesh application can be described as operations over the block. Within an application code, this corresponds to loops over a given block, accessing data through a stencil, performing some calculations, then writing back (again through the stencils) to the data arrays. Consider the `ideal_gas` loop from CloverLeaf’s reference implementation [2] as detailed in Figure 3, operating over each grid point in the structured mesh. Note that here the data arrays are all declared as Fortran allocatable 2D arrays. The loop operates in column major order.

```c
DO k=y_min,y_max
  DO j=x_min,x_max
    v=1.0/density(j,k)
    pressure(j,k)=(1.4-1.0)*density(j,k)*energy(j,k)
    pressurebyenergy=(1.4-1.0)*density(j,k)
    pressurebyvolume=-density(j,k)*pressure(j,k)
    sound_speed_squared=v*v*(pressure(j,k)*pressurebyenergy-pressurebyvolume)
    soundspeed(j,k)=SQRT(sound_speed_squared)
  ENDDO
ENDDO
```

**Figure 3:** Original `ideal_gas` loop from CloverLeaf

An application developer declares this loop using the OPS API as illustrated in Figure 4 (lines 29-34), together with the “elemental” kernel function (lines 1-13). The elemental function is called a “user kernel” in OPS to indicate that it represents a computation specified by the user (i.e. the domain scientist) to
apply to each element (i.e. grid point). By “outlining” the user kernel in this fashion, OPS allows to factor out the declaration of the problem from its parallel implementation as we will show later in Section 3. The macros \texttt{OPS\_ACC0, OPS\_ACC1, OPS\_ACC2} etc.\footnote{A similar approach is used in the C kernel implementations of the original CloverLeaf application} will be resolved to the relevant array index to access the data stored in \texttt{density0, energy0, pressure} etc. The explicit declaration of the stencil (lines 17-19) additionally will allow for error checking of the user code. In this case the stencil consists of a single point referring to the current element. More complicated stencils can be declared giving the relative position from the current \((0,0)\) element. For example a stencil accessing the four near neighbors (top, bottom, left,right) can be declared as \texttt{S2D\_4POINT} in Figure 4. The \texttt{OPS\_READ} indicates that \texttt{density0} will be read only. The \texttt{ops_par_loop} declares the structured block to be iterated over, its dimension, the iteration range and the \texttt{ops\_data}s involved in the computation. The actual parallel implementation of the loop is specific to the parallelization strategy involved. OPS is free to implement this with any optimisations necessary to obtain maximum performance.

In structured mesh applications there are often cases where the iteration range is over a sub-set of the dimensions. Thus for example there are loops over a 2D face of a 3D block accessing both the 3D
dataset and data from a 2D dataset. To declare such an iteration, OPS uses a special type of stencil called a *strided stencil* (see line 25-27 in Figure 4). In this case the additional stride array is set to 1 for the y-dimension and 0 for the x-dimension to indicate that only the y-dimension is iterated over, i.e. a stride over y.

The ops_arg_dat(...) in Figure 4 indicates an argument to the parallel loop that refers to an ops.dat. A similar function ops_arg_gbl() enables users to indicate global reductions. An example is illustrated in Figure 5. In this case the OPS_MIN indicates that the operation is to obtain the global minimum and store it in local_dt.

The final API call of interest concerns the declaration of global constants (ops_decl_const(...)). Global constants require special handling across different hardware platforms such as GPUs. As such OPS allows users to indicate such constants at the application level such that its implementation is tailored to each platform to gain best performance.

### 2.2 Porting CloverLeaf to OPS

The original CloverLeaf 2D application written in Fortran 90 was converted to the OPS API by manually extracting the user kernels, outlining them in header files and converting the application to the OPS's C/C++ API. All effort was taken to keep the naming conventions of routines and files as similar to the original as possible. After conversion, the OPS CloverLeaf version consisted of 80 ops_par_loops spread across 16 files. The application is about 4600 lines of code. The OPS back-end library (implemented in C and C++) which currently supports parallelizing over multi-core CPUs (with OpenMP), distributed memory (MPI) and single GPUs (with CUDA) including common support functions for all these back-ends and other utility functions, plus the code generation tools, in total consists of about 9000 lines of source code.

In comparison each of the original CloverLeaf implementations are self contained separate parallel implementations, one for CUDA+MPI, OpenMP+MPI etc. The CloverLeaf reference implementation (i.e. the MPI+OpenMP parallelization) consists of about 7000 lines of source code.

Once converted to the OPS API, an application can be validated as a single threaded implementation, simply by including the header file ops_seq.h and linking with OPS's sequential back-end library. The header file and the library implement API calls for a single threaded CPU and can be compiled and linked using conventional (platform specific) compilers (e.g. gcc, icc) and executed as a serial application.

The serial developer version allows for the application's scientific results to be inspected before code generation takes place. It also validates the OPS API calls and provides feedback on any errors, such as differences between declared stencils and the corresponding user kernels or differences between data types. All such feedback is intended to reduce the complexity of programming and ease debugging. There is opportunity at this stage to add further checks and tests to increase developer productivity. Including the developer header file and linking with OPS’s distributed memory (MPI) back-end libraries can also be used to obtain a low performance MPI parallelization of the application for testing the application. The full CloverLeaf developer version can be found under the OPS git-hub repository [1].

The manual conversion of the original application was not difficult, but nonetheless results in some loss of developer productivity. There are currently no automated tools for converting an arbitrary legacy application to the OPS abstraction. However once converted, the use of OPS to generate different parallelizations of the application was trivial. Therefore we believe that the conversion is an acceptable one-off cost for legacy applications attempting to utilize the benefits of high level frameworks such as DSLs or Active Libraries. As we will show in this report, the advantages of such frameworks far outweigh the costs, by significantly improving the maintainability of the application source, while making it possible to also gain near optimal performance and performance portability across a wide range of hardware.

### 3 Code Generation with OPS

Once the application developer is satisfied with the validity of the results produced by the sequential application, parallel code can be generated. The build process to obtain a parallel executable as illustrated in Figure 6 follows that of OP2's code generation process. The API calls in the application are parsed by the OPS source-to-source translator which will produce a modified main program and back-end specific code. These are then compiled using a conventional compiler (e.g. gcc, icc, nvcc) and linked against platform specific OPS back-end libraries to generate the final executable. As mentioned before, there is the option to read in the mesh data at runtime. The source-to-source code translator is written in Python and only needs to recognize OPS API calls; it does not need to parse the rest of the code. We have deliberately resorted to using Python and a simple source-to-source translation strategy to significantly simplify the complexity of the code generation tools and to ensure that the software technologies on which it is based has longterm support. The use of Python makes the code generator easily modifiable.
allowing for it to even be maintained internally within an organization. Furthermore, as we will show in the following sections, the code generated through OPS is itself human readable which helps with maintenance and development of new optimisations. OPS currently supports parallel code generation to be executed on (1) single threaded vectorized CPUs, (2) multi-threaded CPUs/SMPs using OpenMP, (2) single NVIDIA GPUs, (3) distributed memory clusters of single threaded CPUs using MPI and (4) a cluster of multi-threaded CPUs using MPI and OpenMP.

Figure 6: OPS code generation and build process

3.1 Optimized Single-threaded CPU Version with Vectorization

As mentioned before, the OPS developer version of CloverLeaf simply includes the ops_seq.h file and links to the sequential back-end library to provide a single threaded CPU implementation. Figure 7 illustrates the developer version’s implementation of the ideal_gas loop in CloverLeaf.

C++ templates are used to provide an overloaded implementation to the function ops_par_loop based on the number of arguments for the loop. ideal_gas has four arguments as ops_args, each referring to an ops_dat holding the data to be used in the elemental computation specified by the function pointer named kernel. The p_a[0], p_a[1], p_a[2] and p_a[3] pointers are set up (not shown here) to point to the beginning of the valid data in each of the ops_dats. The total_range variable holds the total iteration range based on the *range array. The idea is to then loop over the total iteration range, calling the elemental user kernel by passing the relevant pointers to the data and updating the data pointers for the next iteration. The offs[i][m] gives the number of bytes to add to the current pointer for each dimension m in order to get the pointer to the next element. The use of a flat iteration loop implementation is in order for the header file implementation to be usable by applications that operates over meshes with any dimension not just 2D.

It is clear that the developer single threaded implementation is not optimal. The key issue is that all the available knowledge about the application at compile time, such as the dimension of the mesh, and type information of the data is required to obtain further optimisations. Additionally the flat loop iteration is not amenable to auto-vectorization by the compiler. Thus the first code generation attempted with OPS is to generate modified platform specific source code that optimizes the sequential performance of a single CPU core. Figure 8 illustrates the modified application level program and Figure 9 illustrates the platform specific code generated for ideal_gas.

The code generator makes use of the fact that this is a 2D application, operating on a 2D structured mesh and generates a 2D nested loop, one over the y-dimension and the other over the x dimension. The iteration ranges for each dimension is readily available in the *range array. start[0], start[1] and end[0], end[1] are initialized using the start index (inclusive) and the end index (exclusive) of the iteration range for the x and y dimensions respectively. The offsets required to shift the data in each dimension are also computed up front so that a vectorized loop over the x-dimension can be code generated. The macro SIMD_VEC can be used to set the vector length for the specific CPU on which the code will be executed and the #pragma simd forces the compiler to vectorize over that loop. The remainder of the iteration over the x-dimension is completed in a separate loop. For higher dimensional applications, the code generator simply needs to set up the offsets for shifting the pointers in each dimension, and then produce the required number of nested loops to iterate over each dimension. The innermost loop will be vectorized. The use of strided stencils will allow to iterate over a sub-set of the
/* ops_par_loop routine for 4 arguments */

template <class T0,class T1,class T2,class T3>
void ops_par_loop(void (*kernel)(T0*, T1*, T2*, T3*),
char const * name, ops_block block, int dim, int *range,
ops_arg arg0, ops_arg arg1, ops_arg arg2, ops_arg arg3) {
    ...
    ...

    for (int nt=0; nt<total_range; nt++) {
        // call kernel function, passing in pointers to data
        kernel((T0 *)p_a[0], (T1 *)p_a[1], (T2 *)p_a[2], (T3 *)p_a[3]);
        ...
        // figure out the current dimension m
        count[0]--; // decrement counter
        int m = 0; // max dimension with changed index
        while (count[m]==0) {
            count[m] = end[m]-start[m]; // reset counter
            m++; // next dimension
            count[m]--; // decrement counter
        }
        // shift pointers to data
        for (int i=0; i<4; i++) {
            if (args[i].argtype == OPS_ARG_DAT)
                p_a[i] = p_a[i] + (args[i].dat->size * offs[i][m]);
        }
    }
    ...
    ...
}

Figure 7: CloverLeaf’s ideal_gas loop - developer version’s implementation

dimensions. For example to implement boundary conditions on a 3D mesh, we can loop over a 2D face, accessing both the 3D dataset and data from a 2D dataset. Further optimisations for these special cases will be explored in future work.
/* ideal_gas modified loop */
ops_par_loop_ideal_gas_kernel("ideal_gas_kernel", clover_grid, 2, rangexy_inner,
ops_arg_dat(density0, S2D_00, "double", OPS_READ),
ops_arg_dat(energy0, S2D_00, "double", OPS_READ),
ops_arg_dat(pressure, S2D_00, "double", OPS_WRITE),
ops_arg_dat(soundspeed, S2D_00, "double", OPS_WRITE));

Figure 8: CloverLeaf’s ideal_gas loop - modified function call

void ops_par_loop_ideal_gas_kernel(char const *name, ops_block block, int dim, int* range,
ops_arg arg0, ops_arg arg1, ops_arg arg2, ops_arg arg3) {
  ...
  ...
  // set up initial pointers
  ...
  ...
  int n_x;
  for ( int n_y=start[1]; n_y<end[1]; n_y++ ){
    for( n_x=start[0]; n_x<start[0]+((end[0]-start[0])/SIMD_VEC)*SIMD_VEC; n_x+=SIMD_VEC ) {
      // call kernel function, passing in pointers to data - vectorised
      #pragma simd
      for ( int i=0; i<SIMD_VEC; i++ ){
        ideal_gas_kernel( (double *)p_a[0]+ i*1, (double *)p_a[1]+ i*1, (double *)p_a[2]+ i*1,
                         (double *)p_a[3]+ i*1 );
      }
      // shift pointers to data x direction
      p_a[0]= p_a[0] + (dat0 * off0_1)*SIMD_VEC;
      p_a[1]= p_a[1] + (dat1 * off1_1)*SIMD_VEC;
      p_a[3]= p_a[3] + (dat3 * off3_1)*SIMD_VEC;
    }
    for ( n_x=start[0]+((end[0]-start[0])/SIMD_VEC)*SIMD_VEC; n_x<end[0]; n_x++ ){
      // call kernel function, passing in pointers to data - remainder
      ideal_gas_kernel( (double *)p_a[0], (double *)p_a[1], (double *)p_a[2],
                        (double *)p_a[3] );
      // shift pointers to data x direction
      p_a[0]= p_a[0] + (dat0 * off0_1);
      p_a[1]= p_a[1] + (dat1 * off1_1);
      p_a[2]= p_a[2] + (dat2 * off2_1);
      p_a[3]= p_a[3] + (dat3 * off3_1);
    }
    // shift pointers to data y direction
    p_a[0]= p_a[0] + (dat0 * off0_1);
    p_a[1]= p_a[1] + (dat1 * off1_1);
    p_a[2]= p_a[2] + (dat2 * off2_1);
    p_a[3]= p_a[3] + (dat3 * off3_1);
  }
  ...
  ...
}

Figure 9: CloverLeaf’s ideal_gas loop - optimized single threaded implementation
Table 1 presents the single threaded CPU performance on an Intel Sandy bridge CPU core. The CloverLeaf problem consists of a 2D mesh with 960 \times 960 cells and 2955 iterations are required to complete the solution. The OPS optimized version is within 6% of the runtime of the original CloverLeaf implementation on this system, pointing to the fact that the layered high-level abstraction had negligible effect on the application performance.

<table>
<thead>
<tr>
<th>Seq. Version</th>
<th>Run time (seconds)</th>
<th>% runtime difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original reference</td>
<td>885.89</td>
<td>-</td>
</tr>
<tr>
<td>OPS - developer</td>
<td>1892.09</td>
<td>-53.17</td>
</tr>
<tr>
<td>OPS - optimized</td>
<td>940.66</td>
<td>-5.8</td>
</tr>
</tbody>
</table>
3.2 OpenMP Parallelization for Multi-core CPUs

Extending the code generator to utilize multi-threaded parallelism based on OpenMP was the first parallel implementation attempted. OpenMP parallelization required the iteration range that is computed over to be distributed among the available number of threads. In this case we retained the vectorization implementation, while allocating a different iteration range for each thread as illustrated in Figure 10. This multi-threading of the outer loop ensures minimal start/stop thread overhead and no cache contention between threads.

Global arguments were handled by separately allocating a temporary reduction variable for each thread and then combining the results of all the threads (see Figure 11).

The final multi-threaded CPU performance on an Intel Sandy Bridge CPU with 32 cores is given in Table 2. In this case, to reduce the NUMA effects on performance, both the original and OPS OpenMP versions were executed with the KMP_AFINITY environmental variable set to compact. We found that this gave the best performance on this two socket CPU node. The performance of the OPS version in this case, is about 20% faster than the original. We believe that this is due to the setting up of the #pragma omp parallel for with explicit partitioning of the iteration rage and its allocation to specific threads. This is in contrast to utilizing the implicit parallelization produced by a #pragma omp parallel region as used in the original implementation.

<table>
<thead>
<tr>
<th>OpenMP Version</th>
<th>Run time (seconds)</th>
<th>% runtime difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original reference (32 OMP)</td>
<td>135.53</td>
<td>-</td>
</tr>
<tr>
<td>OPS (32 OMP)</td>
<td>106.30</td>
<td>21.57</td>
</tr>
</tbody>
</table>
void ops_par_loop_ideal_gas_kernel(char const *name, ops_block block, int dim, int* range,
  ops_arg arg0, ops_arg arg1, ops_arg arg2, ops_arg arg3) {
...
#ifdef _OPENMP
  int nthreads = omp_get_max_threads( );
#else
  int nthreads = 1;
#endif
...
#pragma omp parallel for
for ( int thr=0; thr<nthreads; thr++ ){
  int y_size = end[1]-start[1];
  int start_i = start[1] + (((y_size-1)/nthreads+1)*thr);
  int finish_i = start[1] + MIN(((y_size-1)/nthreads+1)*(thr+1),y_size);
  //get addresses per thread
  int start0 = start[0];
  int start1 = start[1] + (((y_size-1)/nthreads+1)*thr);
  //set up initial pointers
  ...
  for ( int n_y=start_i; n_y<finish_i; n_y++ ){
    for ( int n_x=start[0]; n_x<start[0]+(end[0]-start[0])/SIMD_VEC; n_x++ ){
      #pragma simd
      for ( int i=0; i<SIMD_VEC; i++ ){
        ideal_gas_kernel((double *)p_a[0]+ i*1,(double *)p_a[1]+ i*1,(double *)p_a[2]+ i*1,
              (double *)p_a[3]+ i*1);
      }
      //shift pointers to data x direction
      ...
  }

  for ( int n_x=start[0]+((end[0]-start[0])/SIMD_VEC)*SIMD_VEC; n_x<end[0]; n_x++ ){
    //call kernel function, passing in pointers to data - remainder
    ideal_gas_kernel((double *)p_a[0],(double *)p_a[1],(double *)p_a[2],(double *)p_a[3]);
    //shift pointers to data x direction
    ...
  }
  //shift pointers to data y direction
  ...
}

Figure 10: CloverLeaf’s ideal_gas loop - optimized OpenMP implementation
```c
void ops_par_loop_calc_dt_kernel_min(char const *name, ops_block block, int dim, int* range,
ops_arg arg0, ops_arg arg1) {
...  
...  
    double*arg1h = (double *)arg1.data;
  
  #ifdef _OPENMP
    int nthreads = omp_get_max_threads();
  #else
    int nthreads = 1;
  #endif

  //allocate and initialize arrays for global reduction
  //assumes a max of 64 threads
  double arg_gbl1[1 * 64 * 64];
  for ( int thr=0; thr<nthreads; thr++){
    for ( int d=0; d<1; d++){
      arg_gbl1[d*64+thr] = INFINITY_double;
    }
  }

  #pragma omp parallel for
  for ( int thr=0; thr<nthreads; thr++){
    ...  
    for ( int n_y=start_i; n_y<finish_i; n_y++ ){
      for ( int n_x=start[0]; n_x<start[0]+(end[0]-start[0])/SIMD_VEC; n_x++ ){
        //call kernel function, passing in pointers to data -vectorised
        calc_dt_kernel_min((double *)p_a[0]+ i*1, &arg_gbl1[64*thr] );
      }
      //shift pointers to data x direction
      p_a[0]= p_a[0] + (dat0 * off0_1)*SIMD_VEC;
      for ( int n_x=start[0]+((end[0]-start[0])/SIMD_VEC)*SIMD_VEC; n_x<end[0]; n_x++ ){
        //call kernel function, passing in pointers to data - remainder
        calc_dt_kernel_min((double *)p_a[0], &arg_gbl1[64*thr] );
        //shift pointers to data x direction
        p_a[0]= p_a[0] + (dat0 * off0_1);
      }
      //shift pointers to data y direction
      p_a[0]= p_a[0] + (dat0 * off0_2);
    }
  }

  // combine reduction data
  for ( int thr=0; thr<nthreads; thr++){
    for ( int d=0; d<1; d++){
      arg0h = MIN(arg0h, arg_gbl1[d*64+thr]);
    }
  }
}
```

Figure 11: CloverLeaf’s `calc_dt_kernel_min` loop - OpenMP implementation of global reductions
3.3 CUDA Implementation for NVIDIA GPUs

The parallelization on GPUs targeted NVIDIA’s CUDA. Again the code generation and file structure followed the design of the OP2 project. Calls to ops_dec_dat allocate the data both on the host CPU and the device (GPU). The idea is to do all the computation on the device and only leave subsidiary functions such as distributed memory halo exchanges on the host.

Such a design attempts to avoid the performance bottlenecks introduced by the PCIe bus that connects modern GPUs to the CPU. The CUDA code generator parses each of the ops_par_loops and generates a *.cu file containing the host code and the device code. This file also includes the user kernel re-generated as a device function. The ideal_gas loop’s CUDA implementation, generated via the OPS code generator is detailed in Figure 12.

```c
//user kernel
__device__ void ideal_gas_kernel( const double *density, const double *energy,
    double *pressure, double *soundspeed) {
  double sound_speed_squared, v, pressurebyenergy, pressurebyvolume;
  v = 1.0 / density[OPS_ACC0(0,0)];
  pressure[OPS_ACC2(0,0)] = (1.4 - 1.0) * density[OPS_ACC0(0,0)] * energy[OPS_ACC1(0,0)];
  pressurebyenergy = (1.4 - 1.0) * density[OPS_ACC0(0,0)];
  pressurebyvolume = -*density[OPS_ACC0(0,0)] * pressure[OPS_ACC2(0,0)];
  sound_speed_squared = v*v*(pressure[OPS_ACC2(0,0)] * pressurebyenergy-pressurebyvolume);
  soundspeed[OPS_ACC3(0,0)] = sqrt(sound_speed_squared);
}

__global__ void ops_ideal_gas_kernel(
    const double* __restrict arg0, const double* __restrict arg1,
    double* __restrict arg2, double* __restrict arg3,
    int size0, int size1 ){
  int idx_y = blockDim.y * blockIdx.y + threadIdx.y;
  int idx_x = blockDim.x * blockIdx.x + threadIdx.x;
  arg0 += idx_x * 1 + idx_y * 1 * xdim0_ideal_gas_kernel;
  arg1 += idx_x * 1 + idx_y * 1 * xdim1_ideal_gas_kernel;
  arg2 += idx_x * 1 + idx_y * 1 * xdim2_ideal_gas_kernel;
  arg3 += idx_x * 1 + idx_y * 1 * xdim3_ideal_gas_kernel;
  if (idx_x < size0 && idx_y < size1) {
    ideal_gas_kernel(arg0, arg1, arg2, arg3);
  }
}

// host stub function
void ops_par_loop_ideal_gas_kernel(char const *name, ops_block Block, int dim, int* range,
    ops_arg arg0, ops_arg arg1, ops_arg arg2, ops_arg arg3) {
  ...
  ...
  int x_size = end[0]-start[0];
  int y_size = end[1]-start[1];
  ...
  dim3 grid( (x_size-1)/OPS_block_size_x+ 1, (y_size-1)/OPS_block_size_y + 1, 1);
  dim3 block(OPS_block_size_x,OPS_block_size_y,1);
  ...
  ops_H_D_exchanges_cuda(args, 4);
  //call kernel wrapper function, passing in pointers to data
  ops_ideal_gas_kernel<<<grid, block >>> ( (double *)p_a[0], (double *)p_a[1],
      (double *)p_a[2], (double *)p_a[3],x_size, y_size);
  ops_set_dirtybit_cuda(args, 4);
  ...
  ...
}
```

Figure 12: CloverLeaf’s ideal_gas loop - CUDA implementation
The key points to note here are as follows: lines 37-38: The host code sets up the CUDA thread blocks and grids based on the fact that this is a 2D application. OPS_block_size_x and OPS_block_size_y are by default set to 16, but can be reset at runtime to gain any performance improvements based on the problem size. line 41: the host code then transfers the required data by calling the host to device copy function provided by the OPS CUDA back-end library. This affectively updates the device copy of the data if it previously has been written to by the host (i.e. using a dirty bit setting). lines 43-45: calls the CUDA kernel declared in lines 13 to 27. This sets up the thread indices based on the available number of threads per block and calls the user kernel. The user kernel has been regenerated here with the __device__ key word added at the beginning to indicate that it is now invoked as a device function.

Special code is also generated to handle global reductions and global constants. Figure 13 demonstrates the implementation of reductions with CUDA. A host pointer to the reduced argument’s data is created (line 26) first. Then the number of CUDA blocks to be reduced over is computed and device memory required for reductions is reallocated to the necessary size (lines 31-37). OPS_reduct_h and OPS_reduct_d are host and device pointers pointing to the memory reserved for the reductions, the reduced argument’s data pointer is set to point to these addresses and initialized (lines 39-44). Finally the amount of shared memory on the GPU required for the reduction is computed and the CUDA kernel call is invoked with the shared memory. The CUDA kernel performs the reduction on each thread-block using the special OPS CUDA reduction function ops_reduction<OPS_MIN>(..). Now each CUDA thread block holds its reduction result, which is later combined on the host to obtain the final global reduction result (lines 56-62).

Finally global constants on the GPU required special handling. They were declared on the GPU’s constant memory and for access via the GPU’s read-only cache. Runtime performance results of CloverLeaf’s CUDA implementations are presented in Table 3. We have identified that the poorer performance of the OPS version is due to not fusing several kernels together to form larger kernels as done in the original CloverLeaf CUDA parallel implementation. As these are implemented as manual hand-tuning, we have refrained from applying similar fusions to the OPS version in order to keep the application level code unmodified based on the need of a single platform specific implementation. However our aim in the future will be to identify loop fusion opportunities at the code generation stage and generate the fused kernels as required for a given platform automatically.

<table>
<thead>
<tr>
<th>CUDA Version</th>
<th>Run time (seconds)</th>
<th>% runtime difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original CUDA</td>
<td>37.59</td>
<td></td>
</tr>
<tr>
<td>OPS</td>
<td>43.73</td>
<td>-16.33</td>
</tr>
</tbody>
</table>

Table 3: CloverLeaf GPU performance - NVIDIA K20c GPU
// user kernel
__device__ void calc_dt_kernel_min(const double* dt_min, double* dt_min_val) {
    *dt_min_val = MIN(*dt_min_val, dt_min[OPS_ACC0(0,0)]);
}

__global__ void ops_calc_dt_kernel_min(
    const double* __restrict arg0, double* __restrict arg1,
    int size0, int size1 ){
    double arg1_l[1];
    for (int d=0; d<1; d++) arg1_l[d] = INFINITY_double;
    int idx_y = blockDim.y * blockIdx.y + threadIdx.y;
    int idx_x = blockDim.x * blockIdx.x + threadIdx.x;
    arg0 += idx_x * 1 + idx_y * 1 * xdim0_calc_dt_kernel_min;
    if (idx_x < size0 && idx_y < size1) {
        calc_dt_kernel_min(arg0, arg1_l);
    }
    for (int d=0; d<1; d++)
        ops_reduction<OPS_MIN>(&arg1[d+blockIdx.x + blockIdx.y*gridDim.x],arg1_l[d]);
}

// host stub function
void ops_par_loop_calc_dt_kernel_min(char const *name, ops_block Block, int dim, int* range,
    ops_arg arg0, ops_arg arg1) {
    ... 
    double *arg1h = (double *)arg1.data;
    dim3 grid( (x_size-1)/OPS_block_size_x+ 1, (y_size-1)/OPS_block_size_y + 1, 1);
    dim3 block(OPS_block_size_x,OPS_block_size_y,1);
    int nblocks = ((x_size-1)/OPS_block_size_x+ 1)*((y_size-1)/OPS_block_size_y + 1);
    int maxblocks = nblocks;
    int reduct_bytes = 0; int reduct_size = 0;
    reduct_bytes += ROUND_UP(maxblocks*1*sizeof(double));
    reduct_size = MAX(reduct_size,sizeof(double)*1);
    reallocReductArrays(reduct_bytes);
    for (int b=0; b<maxblocks; b++)
        for (int d=0; d<1; d++) ((double *)arg1.data)[d+b*1] = INFINITY_double;
    mvReductArraysToDevice(reduct_bytes);
    ... 
    // determine amount of shared memory required
    int nshared = 0;
    int nthread = OPS_block_size_x*OPS_block_size_y;
    nshared = MAX(nshared,sizeof(double)*1);
    nshared = MAX(nshared*nthread,reduct_size*nthread);
    //call kernel wrapper function, passing in pointers to data
    ops_calc_dt_kernel_min<<< grid,block,nshared >>>((double *)p_a[0],(double *)arg1.data_d,x_size,y_size);
    mvReductArraysToHost(reduct_bytes);
    for ( int b=0; b<maxblocks; b++ ){
        for ( int d=0; d<1; d++ ){
            arg1h[d] = MIN(arg1h[d],((double *)arg1.data)[d+b*1]);
        }
    }
    arg1.data = (char *)arg1h;
    ... 
}

Figure 13: CloverLeaf’s calc_dt_kernel_min loop - CUDA implementation
3.4 Distributed-memory Parallelization with MPI

A key aspect of an OPS application is that it can be developed purely from a traditional sequential execution point of view and the developer need not consider any platform specific parallelization. The multi-core (OpenMP) and many-core (CUDA) parallelizations described in the previous sections were all single (shared memory) node implementations. These parallelizations are fine-grained. However, the size of the problem that can be solved on a single node is limited by its compute and main-memory capacity. The compute capacity is important to solve a problem within an acceptable time to solution. The CUDA parallelization is even more restricted on a single GPU’s global memory capacity. Thus it is essential to have a distributed memory based parallel implementation to solve large problems without being limited by the resources on a single CPU or GPU node. The idea then is to layer the more coarse grained distributed memory parallelization on top of the thread level parallelization so that both can be utilized.

In OPS we assume that distributed memory parallelism is implemented across nodes where a node can be a single CPU core, a multi-core CPU (or a SMP node) or a single GPU. Each MPI process will have the same application program executed (i.e. the SPMD execution style). The message passing interface (MPI) is used in the implementation where the global structured mesh is partitioned among a number of MPI tasks (i.e. MPI processes) and import/export halos are constructed on each to communicate partition boundary data between tasks. Thus the starting-point of a distributed memory implementation is the design of how the ops_blocks, ops_dats and other data structures are distributed among the MPI processes.

The OPS design, for a single structured block application, aims to achieve this decomposition after the structured mesh is declared with an ops_decl_block call. By invoking the special API call - ops_partition(...) the developer can indicate to OPS to partition a given ops_block on the available number of MPI processes (i.e. the MPI_COMM_WORLD universe). This call will compute the necessary sub-mesh sizes calculating which portion of the global mesh is to be held on each MPI process and create a Cartesian coordinate system-based MPI universe for the given block. Of course, for any non-MPI parallelizations, ops_partition(...) simply points to a dummy sequential routine that has no effect. This “partition” call may in the future be integrated into the ops_decl_block’s back-end library implementation for each parallelization or may continue to be a distinct function call that gives the application developer a chance to give information to OPS on the type of decomposition suitable for the relevant ops_block. The design will be based on the future needs of the multi-block structured-mesh API.

Any subsequent ops_decl_dat declarations on this partitioned ops_block will indicate to OPS that the data is to be distributed based on the sub-mesh sizes computed for this ops_block. For applications such as CloverLeaf which initializes the data arrays used in the application, simply passing NULL pointers as in Figure 2 will indicate to allocate ops_dats with empty data arrays. If, on the other hand the problem has initial data to be set at the start of the program execution, OPS will provide HDF5 based parallel I/O routines to read in the data from HDF5 files, using the dimension sizes computed for the ops_block. For example, OPS will define a ops_decl_dat_hdf5 similar to ops_decl_dat but with dat replaced by the file from which the data is read using a keyword name.

As part of the back-end implementation of ops_decl_dat, OPS will create the MPI halos for each dat on each process and MPI data types to handle the halo exchanges. The current implementation relies on MPI’s Cartesian coordinates API and as such can be used for parallelizing ops_blocks with any dimensionality.

A call to ops_par_loop in an OPS application executed under MPI will result in triggering a halo exchange if the rules governing the halo exchange are satisfied. Following a halo exchange, the loop will be executed over the local elements of the mesh on each MPI process. After loop computation is performed, depending on the access and argtype of the ops_arg the halos must be marked as “dirty” so that the next iteration of the loop can make the decision to update the halos as required. The rules governing a halo exchange are as follows:

- If ops_arg is an ops_arg_dat, and ops_arg.acc is OPS_READ or OPS_RW and the dirty bit is set, then do halo exchange and clear the dirty bit
- after the loop iterations, if ops_arg is an ops_arg_dat and if ops_arg.acc is OPS_RW, OPS_WRITE or OPS_INC then set dirty bit to 1

A halo exchange is triggered by a call to ops_exchange_halo(...) which is defined in ops_mpi_rt_support.c in OPS’s MPI back-end library. The code generator determines if a halo exchange is required for a given ops_dat based on the above rules while the dirty bit setting is checked in ops_exchange_halo(...). While halos are only exchanged for ops_arg_dats, for ops_arg_gbls with global operations such as reductions,
the code generator will insert the wrapper functions that trigger the necessary MPI collective operations (MPI_Allreduce) for the required data type.

Two further optimizations which are implemented (1) the use of dynamic halo depths and (2) message aggregation. In the first case the idea is to dynamically select the required depth of the halos to be sent/received based on the stencil used in accessing an ops.dat in a loop. We have identified that such a mechanism significantly reduces the time spent in MPI communications. The second optimisation attempts to combine the data to be sent for all ops.dat within a loop into one single message (one for each direction of each dimension). The original CloverLeaf implementation uses this aggregation optimisation. We have managed to implement a similar strategy in a generic form within the OPS library. The results presented in Section 4 includes these optimisations.

Runtime performance results of CloverLeaf’s MPI implementations are presented in Table 4. The MPI processes were bound to a specific core using the numact1 command at runtime to reduce NUMA issues on the two socket CPU node.

Table 4: CloverLeaf MPI performance - Intel Xeon E5-2680 @ 2.70GHz (Sandy bridge) 32 cores

<table>
<thead>
<tr>
<th>MPI Version</th>
<th>Run time (seconds)</th>
<th>% runtime difference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original (32 MPI)</td>
<td>101.4</td>
<td>–</td>
</tr>
<tr>
<td>OPS (32 MPI)</td>
<td>97.99</td>
<td>3.36</td>
</tr>
<tr>
<td>Original (16 MPI x 2 OMP)</td>
<td>104.61</td>
<td>–</td>
</tr>
<tr>
<td>OPS (16 MPI x 2 OMP)</td>
<td>97.87</td>
<td>6.44</td>
</tr>
</tbody>
</table>
4 Performance

In this section we report further on performance and compare the runtimes of OPS CloverLeaf to those of the original applications. Table 5 provides details of the hardware and software specifications of the benchmark systems. The first two systems, Broomway and Ruby are single node systems which we use to benchmark the multi-threaded CPU and GPU performance respectively. The third system is the UK national supercomputing resource - ARCHER [7] which we use to benchmark OPS's distributed memory runtime. In addition to the clover_bm.in input deck, we make use of the clover_bm16.in input deck which executes the application on a larger 3840 \times 3840 mesh.

All three systems consist of Intel CPUs. To be consistent with the compiler flags recommended for gaining accurate results from the original CloverLeaf application, we enforce IEEE floating-point mathematics compliance on each compiler and benchmarks.

Table 5: Benchmark systems

<table>
<thead>
<tr>
<th>System</th>
<th>Broomway</th>
<th>Ruby</th>
<th>ARCHER (Cray XC30)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>2\times 8-core Intel</td>
<td>2\times Tesla K20c</td>
<td>2\times 12-core Intel</td>
</tr>
<tr>
<td>Architecture</td>
<td>Xeon E5-2680 2.70GHz (Sandy bridge)</td>
<td>2\times 8-core Intel Xeon E5-2640</td>
<td>Xeon E5-2697 2.70GHz (Ivy bridge)</td>
</tr>
<tr>
<td>Memory/Node</td>
<td>64GB</td>
<td>5GB/GPU (ECC off), 64GB</td>
<td>64GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Red Hat Enterprise Linux ES release 6</td>
<td>Red Hat Enterprise Linux Server release 6.4 (Santiago) with CUDA 5.0</td>
<td>Cray Aries</td>
</tr>
<tr>
<td>O/S</td>
<td>Intel CC 14.0.0</td>
<td>Intel CC 14.0.2</td>
<td>Cray C Compilers 8.2.1</td>
</tr>
<tr>
<td>Compilers</td>
<td>OpenMPI 1.6.5</td>
<td>OpenMPI 1.6.4</td>
<td>cray-mpich/6.1.1</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-O3 IEEE_FLAGS(^1)</td>
<td>-O3 IEEE_FLAGS(^1)</td>
<td>-O3 -Kieee</td>
</tr>
</tbody>
</table>

\(^1\)On Intel compilers, IEEE_FLAGS=-ipo -no-prec-div -restrict -fno-alias -fp-model strict -fp-model source -prec-div -prec-sqrt

The first set of results illustrates performance on a single node: a dual-socket multi-core CPU and a single GPU. Figure 14 and Figure 15 present times taken by the main hydro iteration loop to solve the 960\times 960 and 3840\times 3840 meshes. The MPI and OpenMP results are from Broomway while the CUDA results are from one of the GPUs in Ruby. We see that on the CPU system for both problems the OPS version executes within 10% of the original implementation’s runtime. The best runtime for the 960\times 960 mesh is achieved using a 16 MPI \times 2 OMP hybrid configuration on OPS, which is about 3% faster than the best runtime achieved with the original (the MPI version). For the larger mesh the OPS version gives better performance for all combinations of MPI and OMP. The best performance in this case is given by the pure MPI version which is about 8% faster than the best runtime of the original (again given by the pure MPI version). For the GPU version however, the original always gives better performance with up to about 16% faster runtimes than OPS. As mentioned before, we need to investigate further optimisations for the OPS's CUDA code generation and back-end to enable loop fusion to match the original performance.
The final set of results explores the parallel scalability of the application. Figure 16 and Figure 17 present the runtimes of the pure MPI version of CloverLeaf on Archer up to 1536 cores. In this benchmark each core was assigned an MPI process. We see that the OPS version closely follows (and later outperforms) the scaling performance of the original. The percentage runtime breakdown illustrates the increasing cost of MPI communications.

Figure 16: CloverLeaf MPI performance on Archer - 960 × 960 mesh

Figure 17: CloverLeaf MPI performance on Archer - 3840 × 3840 mesh

A plan to further improve the performance is currently being considered with the use of non-blocking MPI communication primitives to achieve latency hiding. This would require separating the iteration range into two sections (1) inner iteration range and (2) boundary range that accesses the halo. Then the idea will be to compute over the inner range which does not access any halo elements, while MPI sends/receives are updating the halos. The boundary range will be iterated over once the halos have been updated.
5 Conclusions and Future Work

In this report we detailed research on applying a domain specific high-level abstraction development strategy to a hydro-dynamics benchmark application from AWE. The aim was to investigate performance portability and “future proofing” a key class of HPC applications. The benchmark application, CloverLeaf was re-engineered to use the OPS high-level framework which targets the development of multi-block structured mesh applications. It provides an API for the development of such applications and appears as a classical software library to the developer. However an application once written with the API can be transformed to different parallel implementations through OPS’s code generation tools enabling it to be executed on a range of multi-core, many-core and distributed memory systems.

CloverLeaf, once converted to use OPS, plus all the code generation tools and the OPS back-end library consisted of approximately twice the size of the code base of the original CloverLeaf ref implementation. However, the application and the OPS code base enables the generation of three distinct parallelizations; distributed memory, shared-memory multi-threading and SIMT many-core. In contrast, the original CloverLeaf code consists of separate additional one-off implementations for each of these parallelizations. In one sense, there is significant replication within each implementation from a code maintenance point of view. But more importantly each version incurs a repeat of the development time, thus significantly replicating the effort of the software developer.

Benchmarked performance of the two versions of CloverLeaf showed that the OPS version out-performs the original OpenMP version on shared memory multi-core Intel Sandy-Bridge CPUs, but is about 15%-20% less faster on an NVIDIA K20 GPU. However, we noted that a number of specific loop fusions implemented in the original CUDA version are not implemented under OPS and this is contributing to this disparity. On a Cray XC30 distributed memory system, OPS showed similar scaling performance to the original, performing within 10-15% of its runtime.

Future work will attempt to embed a number of further optimisations to the OPS code generator and back-end libraries. Firstly, latency hiding will be explored for the distributed memory halo exchanges. Strategies to inform the application developer at code generation time of potential loop fusion opportunities and later automate the fusion will also be investigated. This will be important for gaining higher performance on GPUs and CPUs. The next stage of the OPS CloverLeaf work will be the conversion of the 3D version of CloverLeaf to OPS. This will be accompanied by extending the code generator to obtain an OpenCL and an OpenACC version of the application. Finally, more long-term plans as part of the OPS project will include the extension of the API, code generator and back-end to develop multi-block structured mesh applications. The project will also investigate the implementation of communication avoiding algorithms, such as cache blocking through the use of loop chaining [8].

Our experience shows that the development of parallel HPC applications through the careful factorization of a parallel program’s functionality and implementation, using a high-level framework such as OPS is no more time consuming nor difficult than writing a one-off parallel program targeting only a single parallel implementation. However the OPS strategy pays off with a highly maintainable single application source without compromising performance portability on the parallel systems on which it will be executed. It also lays the groundwork for providing support for execution on future parallel systems. We believe such an approach will be an essential paradigm shift for utilizing the ever-increasing complexity of novel hardware and software technologies.

Acknowledgements

This report used the ARCHER UK National Supercomputing Service. The OPS project is funded by the UK Engineering and Physical Sciences Research Council projects EP/K038494/1, EP/K038486/1, EP/K038451/1 and EP/K038567/1 on “Future-proof massively-parallel execution of multi-block applications”. We are thankful to David Beckingusale and Andrew Mallinson at the University of Warwick, Michael Boulton at the University of Bristol, and Wayne Gaudin and Andy Herdman at AWE for their insights into the original CloverLeaf application and its implementation.

References


