High-level Abstractions for Performance, Portability and Continuity of Scientific Software on Future Computing Systems - CloverLeaf 3D

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

February 2015

Abstract

In this report we present research in re-engineering the 3D version of the CloverLeaf hydrodynamics application to utilize the OPS domain specific high-level abstraction framework. This work follows on from our previous work on the 2D version of CloverLeaf, but is aimed at extending our findings to a more representative application. We present (1) the lessons learnt in the conversion of CloverLeaf 3D to utilize OPS, (2) the lessons learnt in the development of two new parallelizations for OPS, namely OpenCL and OpenACC and applying them for parallelizing CloverLeaf 3D and (3) performance of the application on a range of modern parallel systems including multi-core CPUs, NVIDIA GPUs, a Cray XC30 distributed memory CPU system and a Cray XK7 distributed memory GPU system. For the purposes of documentation we also detail the recent optimisations carried out for OPS’s distributed memory parallelization, which includes message aggregation to reduce communication times.

Our results re-confirm the conclusions of our previous report 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. Similar to the 2D application, for CloverLeaf 3D we again see how 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.
## Contents

1 Introduction .......................................................... 3

2 Converting 3D CloverLeaf to OPS ................................ 3

3 Brief Notes on New Parallelizations .............................. 4
   3.1 OpenCL .......................................................... 4
   3.2 OpenACC ........................................................ 4

4 Extended Distributed-memory Parallelization with MPI ...... 4
   4.1 Partitioning .................................................... 4
   4.2 Halo Exchange with Message Aggregation ................. 7

5 Performance ............................................................ 12

6 Conclusions ............................................................ 17
1 Introduction

Developing high performance computing (HPC) applications using a high-level abstractions approach has several key advantages, particularly when developing scientific simulation software for modern parallel platforms [1]. The aim is to decouple 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 without being concerned about how it is to be realized on parallel systems. At the same time, a lower implementation level provides opportunity for parallel programming experts to apply novel, aggressive and platform specific optimizations when implementing the required solution on different parallel systems. As a result, this technique enables 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.

The feasibility of this approach for parallelizing HPC applications at the UK AWE plc., was investigated in our previous project, through the CloverLeaf mini-application [2]. CloverLeaf is a self contained unclassified program that represents the performance characteristics of algorithms of interest. The research illustrated the use of the OPS high-level framework [3] in parallelizing the 2D version of CloverLeaf on single-node many-core and multi-core processor systems. The application was first converted to utilize the OPS API. Then OPS code generation tools were used to generate optimized OpenMP code (targeting multi-core CPUs) and CUDA code (targeting NVIDIA GPUs). Performance results showed that the runtime of CloverLeaf executables generated with OPS was within +/-10% of the runtime of the original hand-tuned version.

In this report we present the work carried out to utilize OPS for parallelizing the 3D version of the application. As part of this work we analyse its performance on a further extended set of software and hardware platforms. More specifically we report on the following:

- We note the ease with which the 3D CloverLeaf application could be re-engineered to use OPS. Firstly due to its structural similarities to the 2D version of the application and secondly due to the design of the OPS framework which easily enabled the extension to handle 3D applications.

- The previous report detailed parallelizations based on AVX Vectorization, OpenMP, CUDA and MPI. In this report we present generated code targeting two new parallelizations : OpenCL and OpenACC including various specific issues that had to be solved to obtain an efficient implementation.

- A number of modifications and extensions were implemented into the OPS’s distributed memory (MPI) back-end library. These include changes to the data partitioning strategy and optimized message send/receives with message aggregation. A full discussion of the design and implementation is included here for future reference.

- We benchmark the performance of the resulting parallel code on a range of single node CPU and GPU systems and two distributed memory systems – the Archer [4] Cray XC30 at EPCC and Titan [5] Cray XK7 at ORNL. 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 where available.

The rest of this report is organized as follows. In Section 2 we discuss in detail the process of code generation and the optimisations carried out for the 3D application. Section 3 will discuss each new parallelization; Section 4 details extensions and optimizations to the MPI back-end to facilitate improved message aggregation; these two sections are best read in conjunction with the source code available at [6]. In Section 5, 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 6 notes future work and concludes the report.

2 Converting 3D CloverLeaf to OPS

Porting the 3D version of CloverLeaf was much easier due to its structural similarities to the 2D version. In many cases kernels needed only to be extended to use 3D stencils, with a few new kernels added, such as in the update_halo routine. The conversion itself took only a few days, thanks to the re-usability of previous code. The developer version that did not use any code generation, but included the op_seq.h header file did not need any modifications. The code generators were updated to generate a further outer loop over the z dimension, maintaining vectorization over elements in the x direction and splitting...
work between OpenMP threads in the z dimension. CUDA thread blocks only handle a single z plane with user-specified dimensions in x and y, the grid size in z is set accordingly to the size of the iteration range in z. With only minor fixes to the MPI back-end we were able to get all back-ends working and performing as expected immediately after the conversion itself was completed.

3 Brief Notes on New Parallelizations

3.1 OpenCL

The OpenCL parallelization is similar to the CUDA parallelization, but a number of issues specific to OpenCL had to be addressed. The first is detecting the available platforms and devices on which the application was invoked to execute. Under OpenCL, the ops_init API call will invoke the openclDeviceInit OPS library call to detect the platforms available (on each MPI process if running under MPI) and the number of devices on each platform. A platform could be a CPU, GPU or Accelerator (e.g. Xeon Phi) while there can be multiple devices of the same type of platform. The OPS_CL_DEVICE variable can be set at runtime to select a device type (0 - CPU, 1 - GPU, 2 - Accelerator). By default the device is selected to be a CPU.

OpenCL kernels are only compiled at runtime. Thus the code generator was modified in two ways to enable this process: (1) generate separate .cl files that contain the user kernel and kernel invocation wrapper, (2) generate host code that invokes the OpenCL compiler at runtime to compile the code in the generated .cl files. Thus, when an ops_par_loop is invoked, under OpenCL, it first calls a buildOpenCLKernels_ function appropriate for the loop (passing in appropriate runtime parameters) to compile the user kernel code.

Unlike CUDA, OpenCL does not support global constants. Thus the value of constants needs to be copied to the device each time the kernels are invoked. Finally under MPI+OpenCL new OpenCL kernels to pack and unpack halo data were required similar to the ones found in the MPI+CUDA back-end.

3.2 OpenACC

The OpenACC parallelization is most similar to the OpenMP parallelization, except that the division of the iteration space between threads is not explicit - a simple nested loop is used with explicitly re-calculated indexing into the array. The OpenACC pragmas are added at this point, using the kernels construct. In front of each loop pragma acc loop independent is specified and if any reductions are present, those are listed as well. OpenACC data clauses are currently not used, the applications is linked against the CUDA back-end and all pointers are passed as deviceptr as defined by CUDA interoperability; this currently limits the generality of OPS’s OpenACC parallelization to NVIDIA architectures. In the future we will be able to handle OpenACC more generally and not tied to NVIDIA.

Due to various compiler issues, one .c file (with all the OpenACC pragmas and the nested loop) and one .cpp file (which sets up the initial pointers, places calls to the OPS back-end and calls the execution implemented in the first file) are generated. OpenACC currently has performance issues in 3D that are being investigated with the assistance of PGI.

4 Extended Distributed-memory Parallelization with MPI

In OPS we assume that distributed memory parallelism is implemented across compute nodes where a node can be a single CPU core, a multi-core CPU (or a SMP node) or a single GPU (or accelerator). Each MPI process will have the same application program executed (i.e. in the Single Program Multiple Data - SPMD execution style). The message passing interface (MPI) is used in the implementation where the global structured mesh data 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 MPI processes.

4.1 Partitioning

In the OPS design the mesh decomposition only occurs after all the blocks are declared with ops_decl_block calls and all the data associated with blocks are declared with ops_decl_dat. By invoking the special API call - ops_partition(), the developer can indicate to OPS to partition the ops_blocks
on the available number of MPI processes (i.e. the MPI_COMM_WORLD universe\(^1\)). This will compute the necessary sub-mesh sizes calculating portions of the global mesh 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, \texttt{ops\_partition()} simply points to a dummy sequential routine that has no effect.

The main stages of the partitioning routine are detailed below:

- **Compute the distribution of blocks among MPI processes**
  The first stage is to determine which MPI processes will be responsible for which \texttt{ops\_block}. The distribution of blocks among MPI processes depends on the total number of processes allocated to the problem \(N_P\) and the number of \texttt{ops\_blocks} in this multi-block application, \(N_{\text{block}}\). Two strategies are implemented here:
    
    (a) If number of \texttt{ops\_blocks} > number of MPI proc : spread the blocks among the processes in a round-robin fashion, currently without considering load balance.
    
    (b) If number of \texttt{ops\_blocks} < number of MPI proc : split each block into an equal number of segments. Number of processes allocated to each \texttt{ops\_block} is then \(N_P/\text{block} = N_P/N_{\text{block}}\)

For CloverLeaf, there is only one \texttt{ops\_block} (\(N_{\text{block}} = 1\)). Thus it partitions according to the second strategy.

The following calculations are then performed to determine the partitioning. These set of calculations are done in parallel by each and every MPI process. Thus, every MPI process obtains a global view of the partitioning of the problem, not just the details of the partitioning for the \texttt{ops\_blocks} allocated to it. Each block is first split into the number of processes allocated to it. Given the dimension of a block to be \(N_{\text{dim}}\), we use \texttt{MPI\_Dims\_create(N\_P/\text{block}, N_{\text{dim}}, \text{pdims})} to achieve a balanced split along different dimensions. \texttt{pdims} returns an integer array of size \(N_{\text{dim}}\), specifying the number of MPI processes in each dimension based on a Cartesian coordinate system.

There are generally a number of \texttt{ops\_dat\_s} defined on a given \texttt{ops\_block}. For CloverLeaf there are 45 \texttt{ops\_dat\_s} defined on one single block. The OPS design considers the size of the largest \texttt{ops\_dat} defined on a \texttt{ops\_block} as the “default” size of that \texttt{ops\_block}. Given the largest size in each dimension and given the processes split in each dimension, we can now determine the size of the sub-block allocated to each process. This calculation will involve computing (1) the Cartesian coordinates of each MPI process and (2) the start (global) index (i.e. displacement) of the grid point allocated to the MPI process in each dimension.

- **Decompose each Block**
  The next stage involves the actual decomposition of each \texttt{ops\_block}. Each MPI process determines which \texttt{ops\_blocks} are allocated to it (i.e. owned). If owned, it uses the \texttt{pdims} array for that block to create a Cartesian coordinate based MPI sub-group with \texttt{MPI\_Cart\_create}. Each MPI process then uses \texttt{MPI\_Cart\_coords} and \texttt{MPI\_Cart\_shift} to discover its neighbouring MPI processes within this new MPI group. Finally, it stores away the sub-grid dimensions and displacements computed in the previous stage for future reference. Note that in this stage, all the results from the operations are held locally on each MPI process.

As there is only one \texttt{ops\_block} for CloverLeaf, there is only one MPI group (i.e. all the allocated MPI processes). The single block will be partitioned among the these MPI processes forming a 3D Cartesian decomposition of sub-blocks.

- **Decompose Data defined on a block**
  Each \texttt{ops\_dat}’s local sizes are now determined using the sub-block sizes computed for the \texttt{ops\_block} on which it is defined. The local sizes of a \texttt{ops\_dat} will include the intra-sub-block halos (\texttt{d\_im} and \texttt{d\_ip}) for communicating between MPI processes. For a given dimension, intra-sub-block halos only exists between the sub-blocks that do not hold any boundary grid points in that dimension. However to account for a multi-block design, OPS now considers a block halo (\texttt{d\_m} and \texttt{d\_p}) to exist around each

\(^1\)In the future this will be generalized so that a subset of the MPI_COMM_WORLD as a different communicator can be accepted to support features such as sliding planes where one MPI universe is assigned to OPS and another is assigned for other parallel computations
ops_block. Each ops_dat defined on an ops_block will therefore be allocated additional memory to hold these block halos. The relationship of block-halos and intra-sub-block halos are illustrated in Figure 1: the key difference is that the first is explicitly defined and managed by the user, while the latter is implicitly created and managed by OPS. The global size of an ops_dat including the ops_block halos are computed as zero_base_gbl_size by adding the start base to the global size of the ops_dat and by removing the size of block halos depths at the start (d_m) and end (d_p) of each dimension of the block.

Figure 1: Block-halos and intra-sub-block halos (currently d_im = d_m and d_ip = d_p)

The local size of an ops_dat (i.e. after decomposing it among the MPI processes) is computed by adding the intra-sub-block halos. The intra-sub-block halos are currently taken to have the same depth as the block halos depths d_m and d_p. However, we plan to accurately compute these depths either at code generation time, by inspecting all the stencils that are subsequently used to access this ops_dat or by way of lazy execution techniques. The computed local size is used for allocating the required memory on the local MPI process as well as any device (e.g. GPU or accelerator) attached to it. This size can also be used to read in the portion of data from a file holding initial data. The file I/O capability is not yet supported by OPS, but will be similar to the parallel HDF5 file I/O implemented in OP2. Finally we populate a struct duplicating information in MPI_Datatype for (strided) halo access. Originally MPI_Datatypes were used in this place, but we discovered that a more optimal execution could be achieved with a manual implementation. Finally memory is allocated for the holding send/receive buffers, one for each direction (negative and positive) of a dimension. For a multi-block application a further step will be taken to decompose the block-halos. However for single-block applications such as CloverLeaf this step has no effect.

One final aspect to note is the computation of the prod[] array which holds the cumulative product of the size of the ops_dat in each dimension for various future needs. For a given dimension d, prod[d] = prod[d-1] * size[d], where size[d] is the size of the ops_dat in dimension d on this MPI process. The special value prod[-1] is set to 1.
4.2 Halo Exchange with Message Aggregation

A call to \texttt{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. The main steps in carrying out the halo exchanges are detailed in Figure 2. Each MPI process carries out these steps to determine if it should do a halo exchange and if so what elements are to be sent/received. A halo exchange is only considered for \texttt{ops_arg_dat}s that are accessed as \texttt{OPS_READ} or \texttt{OPS_RW} (i.e. read only or read/write). Additionally the \texttt{ops_arg_dat} should not be an optional argument to the loop. Any \texttt{ops_arg_dat} that holds an “edge dat” will also not require a halo exchange. An “edge dat” in this case is an \texttt{ops_dat} that has a size of 1 in at least one of its dimensions. Such \texttt{ops_dat}s are not partitioned along the dimension where the size is 1 and it is replicated on each MPI process along this dimension. Edge dats are essentially special \texttt{ops_dat} introduced for CloverLeaf’s initialization routines.

Next, we determine whether the iteration range of the loop is over elements held within this local MPI process. If the iteration range is outside of the elements held within a process, then here is no need for this local MPI process to do any halo exchanges. However, we need to be careful as to not exclude MPI processes which do not have any elements to iterate over, but owe halos to neighbouring processes that do have elements to iterate over. This can be computed by calculating the intersection of the loop iteration rage with the full range of elements held within this sub-block (i.e. the sub block held within this local MPI process). In Figure 2: line 10, adding MAX_DEPTH to the iteration rage gives an extended iteration range that includes the maximum possible halo depths.

```
for (each dimension) {
  for (each arg) {
    if (arg.argtype != OPS_ARG_DAT || (arg.acc == OPS_READ || arg.acc == OPS_RW) ||
      arg.opt == 0) { continue } // go to next arg
    if (this is an edge dat) { continue } // no halos for this arg, go to next arg
    other_dims = 1 //intersection with other dims
    for (every other dimension) {
      other_dims = other_dims && ( size in dimension == 1 ||
        ((loop iteration range + MAX_DEPTH) intersection (full sub-block range)))
    }
    if (other_dims == 0) { break } // no halo exchange for this arg
    compute d_neg = MAX(halo depth to send/receive in negative direction)
    compute d_pos = MAX(halo depth to send/receive in positive direction)
    if (d_pos > 0 || d_neg < 0) // halo depth is non-zero
      ops_exchange_halo_packer(); // pack halos to buffers
  }
  Send/Receive halos // start non-blocking MPI comms
  for (each arg) {
    if (arg.argtype != OPS_ARG_DAT || (arg.acc == OPS_READ || arg.acc == OPS_RW) ||
      arg.opt == 0) { continue } // go to next arg
    compute d_neg = MAX(halo depth to send/receive in negative direction)
    compute d_pos = MAX(halo depth to send/receive in positive direction)
    if (d_pos > 0 || d_neg < 0) // halo depth is non-zero
      ops_exchange_halo_unpacker() // unpack halos from buffers
  }
  MPI_Waitall() // wait for non-blocking MPI comms to end
}```

Figure 2: Main steps in carrying out the halo exchanges

Consider the scenario in Figure 3 for a 2D block, decomposed onto four MPI processes with a maximum halo depth of MAX_DEPTH. Let the extended iteration range (i.e. range including MAX_DEPTH in both directions) be from \((X_s, Y_s)\) to \((X_e, Y_e)\). However only MPI process 0 and 1 hold elements in this iteration range. If we assume the full sub-block range of each MPI process to be given by \((x_{i,s},y_{i,s})\) to \((x_{i,e},y_{i,e})\) where \(i\) is the process rank 0,1,2 and 3 in this case, then for each dimension the intersection of the loop iteration range plus MAX_DEPTH with the full sub-block range on each process can be calculated as follows:
Proc 0: \((x_{0,s} \text{ to } x_{0,e}) \cap (X_s \text{ to } X_e)! = 0\) and \((y_{0,s} \text{ to } y_{0,e}) \cap (Y_s \text{ to } Y_e)! = 0\)

Proc 1: \((x_{1,s} \text{ to } x_{1,e}) \cap (X_s \text{ to } X_e)! = 0\) and \((y_{1,s} \text{ to } y_{1,e}) \cap (Y_s \text{ to } Y_e)! = 0\)

Proc 2: \((x_{2,s} \text{ to } x_{2,e}) \cap (X_s \text{ to } X_e)! = 0\) and \((y_{2,s} \text{ to } y_{2,e}) \cap (Y_s \text{ to } Y_e) = 0\)

Proc 3: \((x_{3,s} \text{ to } x_{3,e}) \cap (X_s \text{ to } X_e)! = 0\) and \((y_{3,s} \text{ to } y_{3,e}) \cap (Y_s \text{ to } Y_e) = 0\)

Only process 0 and 1 has a non-zero intersection in both dimensions. Thus, only processes 0 and 1 will carry out the halo exchange.

The maximum halo depths \((d_{\text{pos}}, d_{\text{neg}})\) based on the stencil of each arg_dat are computed next. For example an ops_dat accessed with a 2D stencil of \((-1,0),(1,0),(0,-1)(0,1)\) has \(d_{\text{pos}}[0]=1, d_{\text{pos}}[1]=1\) and \(d_{\text{neg}}[0]=-1, d_{\text{neg}}[1]=-1\). If at least one of these depths are non-zero, then a separate routine, ops_exchange_halo_packer() is called to pack the relevant MPI halos into buffers to be sent and received subsequently. The internal workings of this routine are discussed later. The actual halo transfer occurs next using non-blocking MPI communication routines followed by logic required to match the reception of messages and unpack the received messages. A final call to MPI_Waitall() ensures that all non-blocking communications are complete at the end of this routine.

The ops_exchange_halo_packer() routine, not only assembles/packs the required halos to buffers, but also makes sure that only the minimum number of halos are sent based on (1) the iteration ranges and range of elements held on each process and (2) whether the data has been modified or not. Figure 4 and Figure 5 illustrates the main sections of this routine. The memory buffers which are used for MPI send/receives are held as global memory pointers (line 2). At the end of the ops_exchange_halo_packer() routine, these buffers contain the relevant halos to be sent and received within the main halo exchange routine (at line 21 in Figure 2).

The first step is to recompute the intersection of the loop iteration range plus the positive and negative halo depths (hereafter called the dependency range) with the full sub-block range in each dimension. This is similar to the computation carried out in the main halo exchange routine, except that here the MAX_DEPTH is replaced by \(d_{\text{pos}}\) and \(d_{\text{neg}}\). Next, we compute the send and receive depths in both negative (i.e. left) and positive (i.e. right) directions. The logic for obtaining the depths is detailed in lines 16-53.

Once halo depths are computed according to the dependencies as above, they can be further reduced by examining if they have been modified previously. Figure 5 illustrates this refinement. In this case, each level of halo depth, in each direction has a flag called a dirty-bit which is set to 1 if the halo data at this depth was overwritten. The dirty-bits are set at the end of an ops_par_loop. If the dirty-bits are set for each depth up to an including a particular depth, then a halo exchange will need to send/receive halos up to that depth.

The initial pointer (i2) is set-up to the memory position from where the actual sub-block data to be sent in the negative (i.e. left) direction begins. As OPS holds data in memory as flat 1D arrays, the appropriate initial pointers that corresponds to the start of the halo blocks in each dimension needs to be established. OPS considers the actual data of an ops_dat to begin after any inter-block halos\((d_m,d_p)\)
//global variables to hold the buffers used in halo exchange
ops_buffer_send_1, ops_buffer_recv_1, ops_buffer_send_2, ops_buffer_recv_2

void ops_exchange_halo_packer(
    dat,           //ops_dat to pack
    d_pos,         //halo depth in positive (right) direction
    d_neg,         //halo depth in negative (left) direction
    iter_range,    //loop iteration range in each dimension
    dim,           //current dimension packed
    send_recv_offsets //array holding offsets to data in buffers ) {
//compute the intersection of dependency range with my full range in each dimension
    for (each dimension dim) {
        range_intersect[dim] = (d_neg[dim] + loop iteration range + d_pos[dim])
            intersection (full sub-block range)
    }

    if (d_pos>0) {
        //compute left_send_depth
        if(my left neighbor’s last point is in the iteration range)
            //then it needs full depth required by the stencil
            left_send_depth = d_pos
        else if(iteration range ends somewhere before my range begins)
            //the dependency may still reach into my range from the left
            left_send_depth = MAX(0,d_pos - (start of my range - end of iteration range))
        else left_send_depth = 0
    }

    //compute right_recv_depth
    if(my last point is in the iteration range)
        //then I need full depth required by the stencil
        right_recv_depth = d_pos
    else if(iteration range starts somewhere after my range ends)
        //the dependency may still reach into my range from the right
        right_recv_depth = MAX(0,d_pos - (end of my range - end of iteration range))
    else right_recv_depth = 0

    if (d_neg<0) {
        //compute left_recv_depth
        if(my first point is in the iteration range)
            //I need full depth required by the stencil
            left_recv_depth = -d_neg
        else if (iteration range starts somewhere after my range begins)
            //the dependency may still reach into my left neighbor’s range
            left_recv_depth = MAX(0,-d_neg-(start of iteration range - start of my range))
        else left_recv_depth = 0
    }

    //compute right_send_depth
    if(my right neighbor’s first point is in the iteration range)
        //it needs full depth required by the stencil
        right_send_depth = -d_neg
    else if(iteration range starts somewhere after my right neighbor’s range begins)
        //the dependency may still reach into my range from the right
        right_send_depth = MAX(0,-d_neg-(start of iteration range - end of my range))
    else right_send_depth = 0
}

//decide whether we intersect in all other dimensions
if(no intesection with other dims) return
...

Figure 4: Main steps in ops_exchange_halo_packer() - Part 1

and intra-block (d_im,d_ip) halos. For the given dimension dim, in the negative direction, the initial pointer is at 

\( d_m[dim] \times \text{prod}[dim-1] \)

where \( \text{prod}[dim-1] \) gives the product of elements in all dimensions up to \( dim-1 \). For example in 3D, for the z dimension (i.e. \( dim=3 \)), \( \text{prod}[3-1] \) gives the number of elements in an x-y plane. Note that \( \text{prod} \) was computed at the partitioning stage. Thus effectively the start pointer in the z dimension is at \( \text{prod}[3-1] \times d_m[3] \) grid points after the start of the sub-block. The computation of initial pointers for send and receives in both the negative and positive directions are illustrated in more detail in Figure 6.
What remains to be carried out for packing halos to buffers are (1) compute the send and receive sizes for this dimension (2) allocate enough memory space in the buffers themselves and (3) create an array to hold the offsets (i.e. displacements) from the start of the offsets (i.e. displacements) from the start of the buffer delimiting the start of each halo in each dimension for each \texttt{ops_dat}. As all the \texttt{ops_args} that require a halo exchange in a given \texttt{ops_par_loop} are packed to the same buffer, the use of an array consisting of the start offset for each \texttt{ops_arg} effectively allows OPS to aggregate all halos into one aggregated MPI message for that loop. Finally dirty-bits are cleared to indicate that the halos up to the actual send depth have been updated.

The \texttt{ops_exchange_halo_unpacker()} routine carries out the disassembly/unpacking of the received messages in a given dimension. This is achieved with a similar operation to that of the \texttt{ops_exchange_halo_packer()} routine, but now only concerns itself with computing (1) the initial pointers starting from where the received data is unpacked to (for each direction), (2) receive depths, receive sizes and receive offsets.
A final point that needs further clarification (and documenting) is the computation of the initial pointers in the `ops_exchange_halo_packer()` and `ops_exchange_halo_unpacker()` routines. Figure 6 details the exact computation involved. In Figure 7 we illustrate the positions of these values for a 2D sub-grid of an `ops_dat` held on an MPI process. Note that when receiving from the positive direction the halos are actually received into the memory space at the negative end. In this case, the initial pointer, starting from which the incoming halo is copied into is computed by subtracting the `actual_depth_recv` (i.e. `adr` in Figure) from the halo depth `d_m`. Similarly, halos received from the negative direction are actually received into the memory space at the positive end. However in this case the initial pointer is always at `d_p` regardless of the value of `actual_depth_recv`.
5 Performance

In this section we report on performance and compare the runtimes of OPS CloverLeaf to those of the original applications. Table 1, and Table 2 provides details of the hardware and software specifications of the benchmark systems. The first three systems, Broomway, Ruby and K80 are single node systems which we use to benchmark the multi-threaded CPU and GPU performance respectively. The final two systems are the Archer Cray XC30 at EPCC [4] and Titan Cray XK7 [5] at ORNL. We make use of the clover_bm.in input deck which executes the application on a $96^3$ mesh for the single node benchmarks, $640^3$ mesh for the strong-scaling runs and $256^3$ per socket for the weak-scaling runs on the Archer and Titan systems.

Table 1: Single Node (CPU) Benchmark systems

<table>
<thead>
<tr>
<th>System</th>
<th>Broomway</th>
<th>Ruby</th>
<th>K80</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>2×8-core (32 SMT) Intel</td>
<td>2×Tesla k20c + 4×Tesla k80</td>
<td></td>
</tr>
<tr>
<td>Architecture</td>
<td>Xeon E5-2680 2.70GHz (Sandy bridge)</td>
<td>2×8-core</td>
<td>Intel Xeon E5-2640 2.50GHz (Sandy bridge)</td>
</tr>
<tr>
<td>Memory/Node</td>
<td>64GB RAM</td>
<td>5GB/GPU (ECC off), 64GB RAM</td>
<td>2x12GB per 2 GPUs (ECC off), 128GB RAM</td>
</tr>
<tr>
<td>Interconnect</td>
<td>-</td>
<td>-</td>
<td>FDR Infiniband (not used)</td>
</tr>
<tr>
<td>O/S</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>CentOS 6.6</td>
</tr>
<tr>
<td>Compilers</td>
<td>Intel CC 14.0.0 OpenMPI 1.6.5</td>
<td>Intel CC 14.0.2 OpenMPI 1.6.4</td>
<td>Intel CC 15.0</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-O3 IEEE_FLAGS(^1) -gencode arch=compute_35, code=sm_35</td>
<td>-O3 IEEE_FLAGS(^1) -gencode arch=compute_35, code=sm_35</td>
<td>-O3 IEEE_FLAGS(^1) -gencode arch=compute_35, code=sm_35</td>
</tr>
</tbody>
</table>

Table 2: Distributed memory Benchmark systems

<table>
<thead>
<tr>
<th>System</th>
<th>Archer (Cray XC30)</th>
<th>Titan (Cray XK7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Architecture</td>
<td>2×12-core Intel Xeon E5-2697 2.70GHz (Ivy Bridge) + NVIDIA K20X</td>
<td>16-core AMD Opteron 6274</td>
</tr>
<tr>
<td>Memory per Node</td>
<td>64GB</td>
<td>32GB + 6GB/GPU (ECC on)</td>
</tr>
<tr>
<td>Interconnect</td>
<td>Cray Aries</td>
<td>Cray Gemini</td>
</tr>
<tr>
<td>OS</td>
<td>CLE</td>
<td>CLE</td>
</tr>
<tr>
<td>Compilers and Flags</td>
<td>Cray C Compilers 8.2.1 cray-mpich/6.1.1 -O3 -Klee</td>
<td>Cray C Compilers 8.2.2 -cray-mpich/6.3.0 -O3 -hgnu -O3 -arch=sm_35</td>
</tr>
<tr>
<td>PGI Compiler</td>
<td>13.10-0</td>
<td></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 gives the performance on single node systems. Figure 8 presents performance on multi-core CPUs based on the Intel x86_64 architecture. Figure 9 and 10 details results on the NVIDIA K20 and K80 GPUs. They illustrate the times taken by the main hydro iteration loop to solve a 96$^3$ mesh. We see that on the CPU system the OPS version executes within 10% of the original implementation’s runtime. In case of OpenMP the OPS version gives a better performance than the original on the Intel CPU system. On this system, the best runtime for the 96$^3$ mesh is achieved using a 32 MPI procs configuration with OPS, which is about 4% faster than the best runtime achieved with the original. In this case the MPI version gives the fastest runtime.

Only a hand-ported OpenCL version of the original 3D CloverLeaf application was available. In this case its run-times on the NVIDIA K20c GPUs gave better performance than OPS’s code generated version by about 14%. However, this gain we believe is due to various hand-optimizations such as manual loop fusions that has significantly changed the reference implementation. On the K80, we could not run the original CloverLeaf 3D application under OpenCL due to the application requiring newer OpenCL libraries than what was available on the system. A single K80 GPU card holds two physical GPUs. Thus the total system was able to execute an 8 way MPI+CUDA job.
The final set of results explores the parallel scalability of the application. Figure 11 and Figure 12 presents the strong-scaling runtimes of the pure MPI and MPI+OpenMP versions of CloverLeaf 3D on the Archer (Cray XC30) system, on up to 12K cores respectively. The total problem size solved is a mesh of size $640^3$. This is a mesh with over 260 million cells. For pure MPI runs, each core was assigned an MPI process. For MPI+OpenMP runs each node was assigned 12 MPI processes each running 2 OpenMP threads. After some experimentation on a single node, this configuration was determined to be the best performing combination of MPI procs and OMP threads. In this case, 2 OpenMP threads are guaranteed to be on the same processor socket. We see that in both parallelizations the OPS version closely follows (and later outperforms) the scaling performance of the original.
The related weak-scaling runs are given in Figure 13 and Figure 14. In this case we assign a mesh of $256^3$ cells per socket. Thus, for the largest execution done on Archer the total problem size is just over 17 Billion cells. Again we see that the OPS version closely follows the original and in fact gives about 10% better performance throughout each of the runs.
Executing both of the above problems on the ORNL’s Titan (Cray XK7) system allows us to add the performance of OPS’s MPI+CUDA parallelization. Strong-scaling results are given in Figure 15 and Figure 16. Weak-scaling results are given in Figure 17 and Figure 18. The performance profiles on Titan are also similar to that of Archer for MPI and MPI+OpenMP parallelizations, where the OPS version closely follows the original. There was no direct port of the 3D application to CUDA and thus only the OPS MPI+CUDA version was benchmarked on Titan.

Figure 15: CloverLeaf3D Strong-scaling - MPI $640^3$ mesh on Titan (Cray XK7)

Figure 16: CloverLeaf3D Strong-scaling - MPI+OMP and MPI+CUDA $640^3$ mesh on Titan (Cray XK7)

Figure 17: CloverLeaf3D Weak-scaling - MPI $256^3$ mesh per socket on Titan (Cray XK7)

Figure 18: CloverLeaf3D Weak-scaling - MPI+OMP and MPI+CUDA $256^3$ mesh socket on Titan (Cray XK7)
6 Conclusions

In this report we detailed research in re-engineering the 3D version of the CloverLeaf hydrodynamics application to utilize the OPS domain specific high-level abstraction framework. Additionally we noted lessons learnt in developing two new parallelizations that can be developed with OPS, namely OpenCL and OpenACC and recent optimisations carried-out for OPS’s distributed memory parallelization, which includes message aggregation to reduce communication times.

The 3D application’s performance was benchmarked on a range of parallel multi-core and many-core systems. Where available, the original 3D CloverLeaf application’s performance was also benchmarked on these systems and compared to that of the related OPS parallel version.

We see that on the CPU system, using MPI and/or OpenMP parallelizations the OPS version executes within 10% of the original implementation’s runtime. The original CloverLeaf 3D application parallelized with OpenCL on the NVIDIA GPUs gave better performance than OPS’s code generated version by about 14%. However, this gain we believe is due to various hand-optimizations such as manual loop fusions that has significantly changed the reference implementation.

The final set of results explored the parallel scalability of the application both strong-scaling and weak-scaling on the Archer (Cray XC30) distributed memory CPU cluster (on up to 12K cores) and the Titan (Cray XK7) GPU cluster (on up to 1024 GPUs). For MPI only, MPI+OpenMP and MPI+CUDA parallel runs on these systems, we see that the OPS version closely follows (and at large machine scale outperforms) the scaling performance of the original.

The above results re-confirm the conclusions of our previous report 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. Similar to the 2D application, for CloverLeaf 3D we again see how 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.

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 project will also investigate the implementation of communication avoiding algorithms, such as cache blocking through the use of loop chaining [7].
Acknowledgements

This research is funded by the UK AWE plc. under project “High-level Abstractions for Performance, Portability and Continuity of Scientific Software on Future Computing Systems”. 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” and EP/J010553/1 “Software for Emerging Architectures” (ASEArch) project. This paper used the Archer UK National Supercomputing Service from time allocated through UK Engineering and Physical Sciences Research Council projects EP/I006079/1, EP/I00677X/1 on “Multi-layered Abstractions for PDEs”. This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. CloverLeaf development is supported by the UK Atomic Weapons Establishment under grants CDK0660 (The Production of Predictive Models for Future Computing Requirements) and CDK0724 (AWE Technical Outreach Program) and also the Royal Society through their Industry Fellowship Scheme (IF090020/AM). We are thankful to Endre László at PPKE Hungary for his contributions to OPS, David Beckingsale at the University of Warwick and Michael Boulton at the University of Bristol for their insights into the original CloverLeaf application and its implementation.

References


