An MPI Halo-Cell Implementation for Zero-Copy Abstraction

Jean-Baptiste Besnard
ParaTools SAS
Bruyères-le-Châtel, FRANCE
jbbesnard@paratools.com

Allen Malony
ParaTools Inc.
Eugene, USA
malony@paratools.com

Sameer Shende
ParaTools SAS
Bruyères-le-Châtel, FRANCE
sameer@paratools.com

Marc Pérache
CEA, DAM, DIF
F91297 Arpajon, FRANCE
marc.perache@cea.fr

Patrick Carribault
CEA, DAM, DIF
F91297 Arpajon, FRANCE
patrick.carribault@cea.fr

Julien Jaeger
CEA, DAM, DIF
F91297 Arpajon, FRANCE
julien.jaeger@cea.fr

ABSTRACT

In the race for Exascale, the advent of many-core processors will bring a shift in parallel computing architectures to systems of much higher concurrency, but with a relatively smaller memory per thread. This shift raises concerns for the adaptability of HPC software, for the current generation to the brave new world. In this paper, we study domain splitting on an increasing number of memory areas as an example problem where negative performance impact on computation could arise. We identify the specific parameters that drive scalability for this problem, and then model the halo-cell ratio on common mesh topologies to study the memory and communication implications. Such analysis argues for the use of shared-memory parallelism, such as with OpenMP, to address the performance problems that could occur. In contrast, we propose an original solution based entirely on MPI programming semantics, while providing the performance advantages of hybrid parallel programming. Our solution transparently replaces halo-cells transfers with pointer exchanges when MPI tasks are running on the same node, effectively removing memory copies. The results we present demonstrate gains in terms of memory and computation time on Xeon Phi (compared to OpenMP-only and MPI-only) using a representative domain decomposition benchmark.

Keywords

MPI, Ghost-Cells, Zero-Copy, memory, MPI_Halo

1. INTRODUCTION

As supercomputer architectures steadily increase the number of processing cores, applications developers must adapt their code to meet new requirements of greater concurrency and smaller memory availability per core (hence, per thread). While power dissipation has constrained the processor clock frequencies, manycore devices, such as the Intel Xeon Phi "Knight-Corner" with more than 60 cores and four threads per core, will require several hundred threads to gain full performance potential. In this context, applications are facing a real challenge, being forced to express more parallelism, not only to improve, but also to preserve their current level of performance due to lower frequencies. For many applications, the computation must be decomposed across a larger number of processing units while limiting memory and communication overhead. Unfortunately, maintaining the MPI programming paradigm under these pressures is difficult. Many developers have resorted instead to hybrid methods, adopting shared memory parallelism at the node level. Rewriting applications to use hybrid methods can be problematic.

In this paper, we study the impact of increasing core count and decreasing memory per core by looking at simulations that split a data domain over several distributed computation units. The common way of addressing such problem is to rely on *halo-cells* (or *ghost-cells*) whose role is to locally replicate the domain residing in remote memories. While satisfying computational dependencies, ghost-cells rely on buffers to mirror remote data, and communications to update state. To analyze the *domain decomposition* scenario, we start from the well-known speedup (S) equation, considering sequential (seq), parallel (par) and communication ($comm$) times for a problem of size n on p processing units:

$$S(n,p) = \frac{seq(n)}{par(n,p)} = \frac{seq(n)}{\frac{seq(n)}{p} + comm(n,p)} \quad (1)$$

This simple equation summarizes the scalability challenge. For a very large p, the speedup for a constant problem size (strong scaling) is bounded by communications. If the overhead of communications is canceled, we reach the ideal speedup of p. However, when problem size increases (weak scaling), scalability is bounded by the amount of memory available. This can be illustrated by modeling *iso-time* execution for a linearly growing problem size P_s. Starting from $P_s = sp$, where s is the memory available per core, we argue that there will be a parallel overhead requiring a problem size reduction of s_{comm} in order to be compensated, defining $C(s_{comm})$ as the time to compute a size s_{comm}. More generally, in order to allow weak-scaling, s_{comm} must have a

EuroMPI '15, September 21-23, 2015, Bordeaux , France

DOI: http://dx.doi.org/10.1145/2802658.2802669

lower complexity than the problem size, $\Theta(sp)$, being therefore at most linear, such as $P_s = (s - s_{comm})p$ with $C(s_{comm})$ compensating an overhead independent from p.

It is interesting to see that complex computation requires less compensation. Consequently, regular domain decomposition involving heavy computation with a constant number of neighbors is a good candidate for weak-scaling. We chose this scenario to study for this reason. Considering common domain decomposition approaches, we first model the cost associated with halo cells, including hybrid programming as a (compulsory) avenue on current hardware. Then, we develop a compact MPI abstraction, called `MPI_Halo`, that simplifies communication expression, while transparently providing shared-memory programming advantages to programs relying solely on the MPI model. `MPI_Halo` has the advantage of being close to the MPI semantics with minimal code modification, while providing the performance characteristics of hybrid computation. Eventually, after describing related work, the paper concludes with prospects for future work.

2. THE HALO-CELL MODEL

We consider the problem of modeling the number of halo-cells for common topological decompositions to illustrate the challenges associated with scaling domain decomposition problems to a large number of threads. In particular, we show that the model argues for a shift from pure MPI processes to hybrid parallelism when considering memory constraints. This analysis motivates the contribution of the `MPI_Halo` interface as a means to semantically hide hybrid programming disruption, while retaining its performance.

To model halo-cells, we study classical mesh-based decomposition in various dimensions. We base our model on wrapped-around meshes which have a fixed degree for each node: two in one dimension (1D) (Figure 1(d)), eight in two dimensions (2D) (Figure 1(e)) and twenty-seven in three dimensions (3D) (Figure 1(f)). This property simplifies halo-cell analysis while providing a valid approximation for unwrapped meshes (Figures 1(a), 1(b), 1(c)) where only a node subset is part of boundaries. In general, to efficiently scale a problem, the parallelism overhead should remain negligible when compared to the actual computation. Therefore, in the case of domain splitting, a good scaling is linked to the reduction of the number of halo-cells, which impact computation in two ways: (1) communications and (2) domain duplication (i.e., buffering).

The goal of our modeling for wrapped-around topologies is to compute the number of cells needed to achieve a given halo-cell ratio. Figure 2 introduces the method we used to compute the number of halo-cells. It relies on a characteristic length l which can be seen as the "side" of the mesh in function of its dimension. For example, considering a two-dimensional mesh gathering n cells, $l = \sqrt{n}$. For d dimensions, $l(n,d) = n^{\frac{1}{d}}$. Thus, it is trivial to compute the size of the mesh with ghost cells starting from a characteristic length increased to $2C$ with C the number of ghost cell layers (illustrated in red in Figure 2). Then, ghost cell count N_g is computed by subtracting the size without halo-cells from the size with halo-cells – yielding the following equation for wrapped-around meshes:

$$N_g(n,d) = (l(n,d) + 2C)^d - l(n,d)^d = (n^{\frac{1}{d}} + 2C)^d - n \quad (2)$$

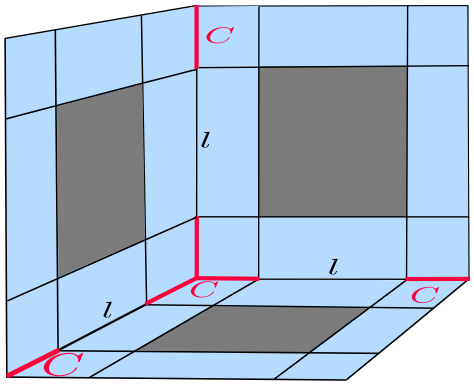


Figure 2: Deriving ghost-cell count from the characteristic length (cube-embedding).

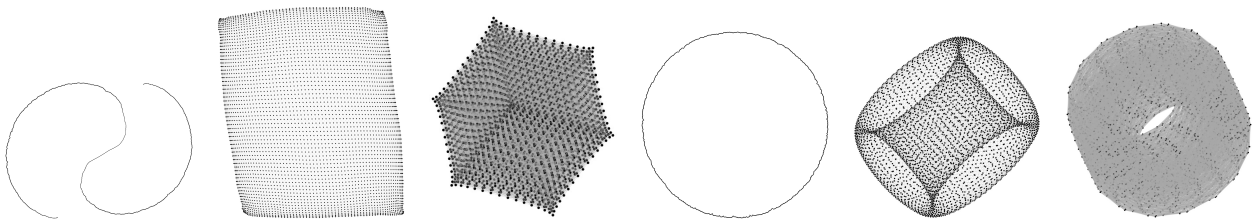
d	1	2	3
$N_g(p,n,C)$	$2pC$	$4pC(\sqrt{\frac{n}{p}} + C)$	$2pC(3\frac{n}{p}^{\frac{2}{3}} + 6C\frac{n}{p}^{\frac{1}{3}} + 4C^2)$

Figure 3: Total number of ghost cells for torus-based topologies in various dimensions considering an even split of the number of cells among processes.

Figure 3 shows how Equation 2 can be used to derive the number of ghost cells as a function of mesh dimension d, problem size n, and halo-cell layers C, considering p processes with a local problem size n_p computed by evenly dividing a global problem size n such as $n_p = \frac{n}{p}$. The 1D-torus is the only topology with a number of ghost cells independent from problem size n. Naturally, when increasing mesh dimension, the number of halo-cells increases rapidly, requiring bigger problem sizes to achieve a given ghost-cell ratio.

This problem can be illustrated by letting $r(p,n,C) = \frac{N_g(p,n,C)}{n}$, and then expressing n in function of this new variable to compute the number of cells needed to achieve a defined ratio. As presented in Figure 4, this ratio has an expression linear in terms of the number of processes as global ratio is obtained by reproducing p times a ratio observed on individual processes. Using these equations, Figure 5 represents the number of ghost cells needed to achieve a given ratio, outlining that as torus dimension increases, achieving lower ratios requires more cells. For example, if we consider a basic 3D splitting, with one layer of ghost cells, 2.18×10^8 cells per memory unit are required to achieve a ratio of 1%. This yield a memory usage of $M(s_c,r,p,C) = s_c(1+r) \times n(r,p,C)$ with s_c the size of a cell, or 1.64 GB per memory area for $s_c = 8$ (size of a double). Compare this value to the memory per core on an Intel Xeon Phi (KNC): 8 GB for 240 threads, or 34 MB per thread. Moreover, this figure is clearly optimistic as applications usually depend on several ghost cell layers with larger cells, not to mention that memory is partially used by the runtime and operating system.

Our analysis here shows that memory requirements associated with new architectures prevent distributed-memory models such as MPI to be directly transposed. Indeed, as demonstrated, subdividing a given mesh in small memory areas leads to a waste of memory and increases communication overhead. For this reason, the "scaled MPI" approach has to be either mixed with another shared-memory model



(a) 1D (b) 2D (c) 3D (d) 1D(W) (e) 2D(W) (f) 3D(W)

Figure 1: Meshes with 4096 nodes, (W) stands for wrapped-around, producing a torus.

	1D	2D	3D
$n(r,p,C)$	$\frac{2pC}{r}$	$\frac{4pC^2}{r^2}(r+2+2\sqrt{1+r})$	$\frac{2pC^3}{r^3}(r^2+3r(1+r)^{\frac{2}{3}}+6r(1+r)^{\frac{1}{3}}$ $+9[r+1+(1+r)^{\frac{2}{3}}+(1+r)^{\frac{1}{3}}])$
$n(1\%,p,1)$	$200p$	1.61×10^5p	2.18×10^8p
$n(1\%,p,2)$	$400p$	6.43×10^5p	1.74×10^9p
$n(1\%,p,3)$	$600p$	1.44×10^6p	5.89×10^9p
$n(10\%,p,1)$	$20p$	$1679p$	2.37×10^5p
$n(10\%,p,2)$	$40p$	$6716p$	1.90×10^6p
$n(10\%,p,3)$	$60p$	$15111p$	6.42×10^6p
$n(50\%,p,1)$	$4p$	$79p$	$2639p$
$n(50\%,p,2)$	$8p$	$317p$	$21177p$
$n(50\%,p,3)$	$16p$	$712p$	$71272p$

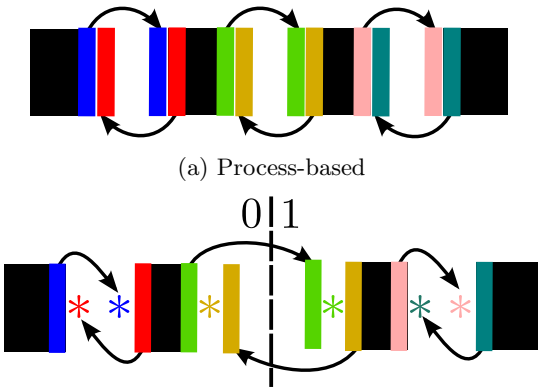
Figure 4: Equations allowing the computation of total number of cells $n(r,p)$ in function of ghost-cell ratio r, number of processes p and the number of ghost cell layers C.

or abstracted in some other way. Section 5 reviews several programming models and approaches addressing this issue. Acknowledging this related research, our work proposes a possible solution that attempts to remain close to the MPI programming paradigm.

3. CONTRIBUTION

As modeled in section 2, domain decomposition on a large number of tasks with reduced memory becomes impractical due to halo-cells memory and communication overhead. In such context, MPI applications have adapted to hybrid model – shared-memory parallelism inside nodes and message-passing between node. Typically, shared-memory parallelism is provided by progressively adding OpenMP constructs to computing loops, interleaving sequential sections between parallel regions. For large numbers of threads, this can result in serious performance inefficiencies.

In this paper, we introduce a way of porting MPI applications to many-core devices based on the MPI halo cells abstraction documented in Figure 6. This interface abstracts halo-cells from their computation, allowing the MPI runtime to avoid copies when data are available in shared memory. This way, an existing MPI-based domain decomposition is transposable to shared-memory with only limited code modification. Of course, this optimization is available only if a subset of remote buffers is directly reachable in shared-memory. One way this could have been done is by gathering mesh cells in a shared-memory segment mapped in MPI processes collocated on the same node, for example thanks to MPI 3.0 shared memory windows. However, we relied on a simpler approach by implementing our `MPI_Halo` interface in a thread-based MPI. We retained the MPC[15] runtime which unifies OpenMP and MPI support for Xeon Phi while providing an integration in both GNU and Intel compilation chains to automatically privatize global variables. Using this approach, our `MPI_Halo` implementation became more flexible – allowing any address to be shared without code modification (recompilation with the correct flag).



(a) Process-based

(b) `MPI_Halo` (two nodes 0 and 1, four processes)

Figure 7: Halo-cell buffer comparison between process-based domain splitting and our node-based `MPI_Halo` approach when considering a 1D splitting.

Figure 7 shows our approach to reduce the number of ghost cell buffers by directly referring to actual data when reachable in shared memory. It compares process-based

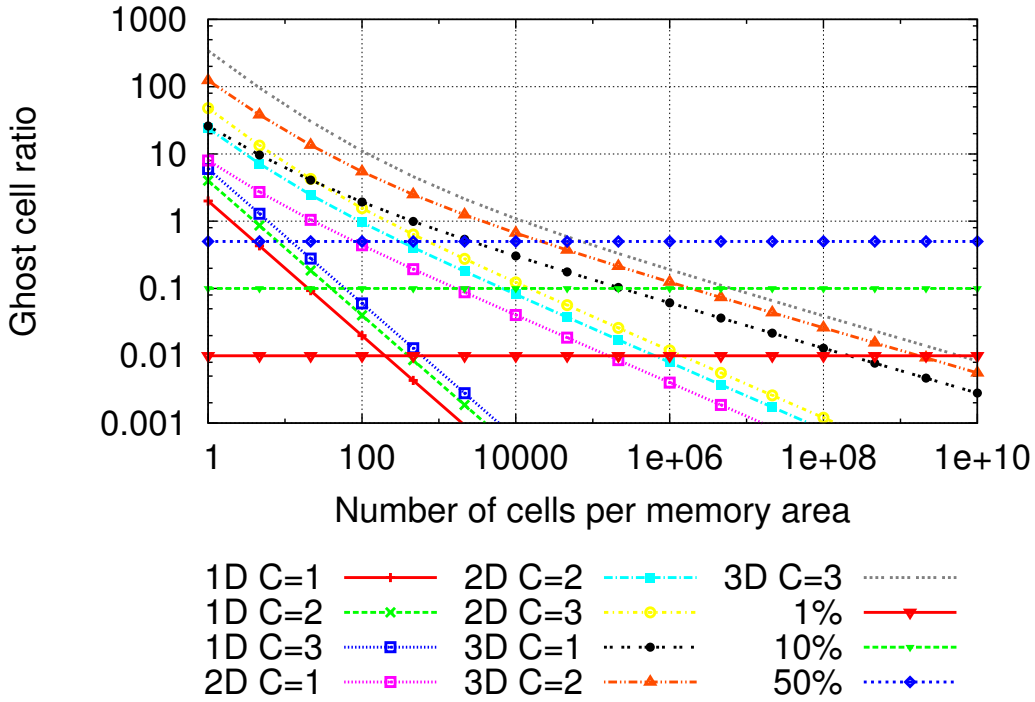


Figure 5: Ghost cell ratio in function of n for various configurations.

(Figure 7(a)) with our node-based `MPI_Halo` approach (see Figure 7(b)), assuming two processes per node). For the latter, only ghost-cells in-between nodes are required. This has the effect of reducing the number of halo-cell buffers, decreasing halo-cell ratio when compared to the classical approach (Figure 7(a)). Naturally, directly accessing remote cells is only possible if data remain valid. However, this kind of dependency is the same as the one arising from local computation where computing a new cell can not be done in place due to spatial dependencies, requiring the allocation of two meshes switched at each time-step, in general. In such configuration, remote cells remain valid during the whole time-step, enabling memory savings. Nonetheless, even if buffer allocation has to be forced due to remote modification, the `MPI_Halo` approach is still advantageous in terms of code readability and communication scheme validation.

To go further, in the thread-based MPI case, tasks located on the same node are running in the same process as threads and therefore share the same address space, allowing our `MPI_Halo` approach to work on any buffer. However, considering an implementation in an MPI 3.0 shared window, only a subset of the address space of each collocated process is visible. In complement, the shared segment does not necessarily start at the same address in each process, preventing the remote pointer to be sent directly. An implementation would therefore have to send only addresses relative to the shared-window start address while requiring memory copies for buffers outside of it (corollary of "outside of node" in the thread-based case). This shows that the `MPI_Halo` mechanism is compatible with MPI 3.0 shared windows and therefore with process-based MPI implementations.

In order to illustrate the advantages of the `MPI_Halo` approach versus point-to-point communications, Figure 8 presents a 1D splitting following Figure 7(b) configuration (retained for conciseness). This code is made of two parts: (1) line 1 to 16 called once in order to initialize the communication scheme, and (2) line 19 to 37 illustrating the use of the `MPI_Halo_ex` handle to trigger communications.

Initialization first sets up four halo cells containers: two for local cells and two for remote cells. During this step, only data-layout and buffer names are provided. Buffers are bound either locally, when buffers are to be sent, or remotely, when buffers are to be received. `MPI_Halo` objects are registered in an `MPI_Halo_ex` handler which abstracts a set of halo-cell exchanges. During this process, remote buffers are queried by name, greatly simplifying the way buffers are associated to each other. At this point, MPI runtime knows both size and dependencies between `MPI_Halo` handlers. At line 16, the `MPI_Halo_ex` object is committed in order to generate the communication scheme based on pairs of send and receive. Internally, this process first relies on a reduction to count the number of incoming requests. Then, each process registered to a remote buffer sends a request which is received by the target using `MPI_ANY_SOURCE`. At this point, the runtime is able to check whether buffers have the same

MPIX_Halo_cell_init(MPI_Halo * h, char * label, MPI_Datatype type, int count) Initializes a halo-cell container with a given data-layout.
MPIX_Halo_exchange_init(MPI_Halo_ex * ex) Initializes an empty `MPI_Halo_ex` object .
MPIX_Halo_cell_bind_local(MPI_Halo_ex ex, MPI_Halo h) Register a halo buffer as local (available for read in remote processes).
MPIX_Halo_cell_bind_remote(MPI_Halo_ex ex, `MPI_Halo` h, int rank, char * label) Register a halo buffer as remote (retrieved from a remote process).
MPIX_Halo_cell_set(MPI_Halo h, void ** ptr) Set the buffer pointer inside the MPI_Halo object (only possible on local `MPI_Halo`).
MPIX_Halo_cell_get(MPI_Halo h, void ** ptr) Retrieve the buffer pointer from the `MPI_Halo` object (only possible on remote `MPI_Halo`).
MPIX_Halo_cell_is_remote(MPI_Halo h) Return true if the `MPI_Halo` required a buffer allocation (only possible on remote `MPI_Halo`).
MPIX_Halo_cell_commit(MPI_Halo_ex ex) Compute the communication scheme associated with bounded halo-cells.
MPIX_Halo_iexchange(MPI_Halo_ex ex) Triggers `MPI_Halo` exchanges in a non-blocking fashion.
MPIX_Halo_iexchange_wait(MPI_Halo_ex ex) Waits for the end of communications associated with this exchange.
MPIX_Halo_exchange(MPI_Halo_ex ex) Triggers `MPI_Halo` exchanges in a blocking fashion.
MPIX_Halo_cell_release(MPI_Halo * h) Releases an `MPI_Halo` object .
MPIX_Halo_exchange_release(MPI_Halo_ex * ex) Releases an `MPI_Halo_ex` object .

Figure 6: Complete `MPI_Halo` interface developed for this paper.

size, type and if the buffer (selected by name) is available locally. A meaningful message is generated in case of error.

In the second part, mesh data are registered in local `MPI_Halo` during computation. At line 25, the asynchronous exchange is initiated, generating pairs of send and receive with independent tags. Only pointers are transparently exchanged instead of complete buffers when data are located on the same node, enabling zero-copy and lowering memory usage. Then, communications are potentially recovered by computation before being waited for in line 27. Eventually, pointers to halo-cells are retrieved from remote `MPI_Halo`, possibly returning a direct pointer if data are located on the same node. Here, zero-copy is possible because we are running MPI tasks in user-level threads thanks to the MPC runtime. If not, `MPI_Halo` will force ghost-cell buffers and fall back to point to point behavior, while maintaining better code readability and error checking.

Dealing with the wait call line 27, in our current implementation it is non-blocking in the sense that it does not require a complete synchronization between all processes. The reason for this is that we have two meshes which are swapped at each time-step, guaranteeing a read only access until next ghost cell exchange with a local synchronization. For other patterns, for example write access, other MPI calls such as MPI_Barrier could be used to outline access epochs, with a potential impact on performance.

In this example, we only presented zero-copy exchanges for contiguous buffers due to limited space. it is also possible to optimize non-contiguous communications. In such cases, local cells would be defined as a derived MPI vector data-type selecting boundary elements and incoming data as a contiguous vector (when relying on a halo-cell buffer). This abstracts the 'packing' phase. However, when data can be reached directly (i.e., via shared-memory) they remain non-contiguous. In such case, computation must be able to process both contiguous buffer (no stride, remote processes) and non-contiguous ones (with a stride, shared-memory). To do so, the `MPI_Halo_cell_is_remote` function can be used to query the kind of buffer being addressed in order to adapt the computing.

Our `MPI_Halo` abstraction allows a compact expression of communications scheme from dependencies between buffers, enhancing readability while providing dynamic checking. Besides, both buffer allocations and copies can be avoided using such abstraction. Clearly, there is the potential for performance gains in respect to both memory and communications. These are discussed below.

4. RESULTS

In this section, we measure the gains provided by our `MPI_Halo` implementation inside the MPC runtime on an Intel Xeon Phi 5110P with 60 cores capable of running 4 threads using 8 GB of memory. Thus, each of the 240 threads has 34 MB of memory, making this device very sensitive to the ghost cell problem we previously outlined. In order to precisely control execution parameters, we decided to write our own benchmark. We chose a image convolution problem for both its simplicity and proximity with other algorithms, like Lattice-Boltzmann which propagates spatial values using similar stencils. Our own code allowed us to benchmark various number of ghost-cell layers, just by changing the convolution kernel. To match the temporal behavior of a simulation code, we ran twenty convolutions, each of them involving halo-cell exchanges.

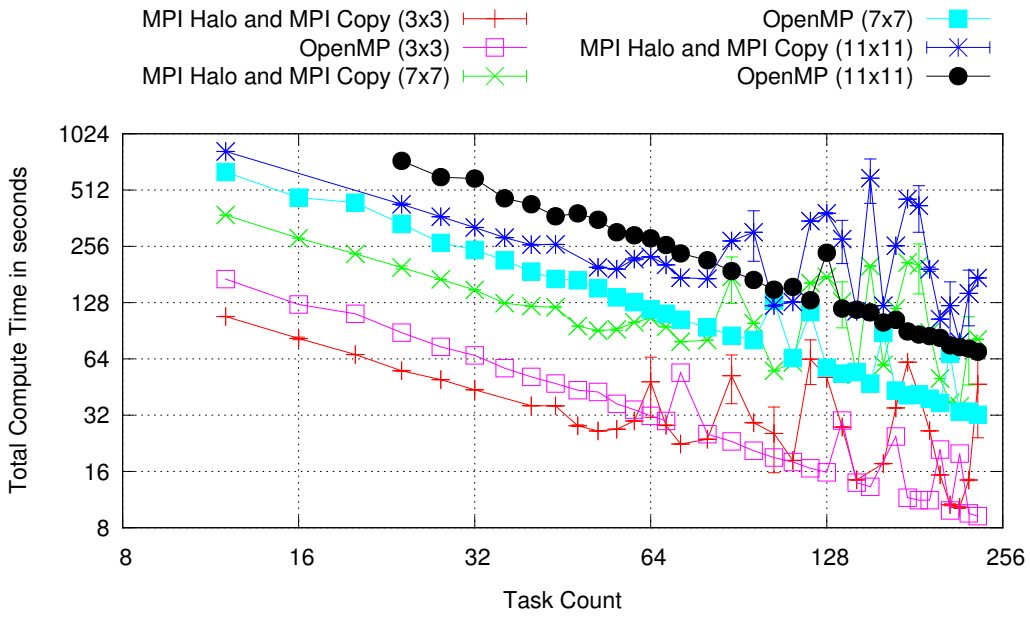
```c
/*------ Initialization (Done once) */
MPI_Halo local_left, local_right, left, right;
/* Name Cells and provide Layout */
MPIX_Halo_cell_init(  &local_left, "Local Left" , MPI_INT, 1024 );
MPIX_Halo_cell_init(  &local_right, "Local Right" , MPI_INT, 1024 );
MPIX_Halo_cell_init(  &left, "Remote Right" , MPI_INT, 1024 );
MPIX_Halo_cell_init(  &right, "Remote Left" , MPI_INT, 1024 );
/* Bind Cells */
MPI_Halo_ex ex;
MPIX_Halo_exchange_init( &ex );
MPIX_Halo_cell_bind_local( ex, local_left );
MPIX_Halo_cell_bind_local( ex, local_right );
MPIX_Halo_cell_bind_remote( ex, right, right_process,  "Local Left" );
MPIX_Halo_cell_bind_remote( ex, left, left_process,  "Local Right" );
/* Generate Communications */
MPIX_Halo_exchange_commit( ex );

/*------ Compute Loop (Called at each time-step)*/
while( compute )
{
  /* Register local cell data */
  MPIX_Halo_cell_set( local_left, mesh  );
  MPIX_Halo_cell_set( local_right, right_coll( mesh )  );
  /* Start asynchronous communications */
  MPIX_Halo_iexchange( ex );
  /* ... Compute mesh center ... */
  MPIX_Halo_iexchange_wait( ex );
  /* Retrieve Ghost arrays */
  int * left_ghost, * right_ghost;
  MPIX_Halo_cell_get( left,  (void **)&left_ghost );
  MPIX_Halo_cell_get( right, (void **)&right_ghost );
  /* ... Compute mesh boundaries ... */
  /* Swap Meshes */
  Mesh * tmp = mesh;
  mesh = oldmesh;
  oldmesh = tmp;
}
```

Figure 8: Illustration of `MPI_Halo` interface usage in C.

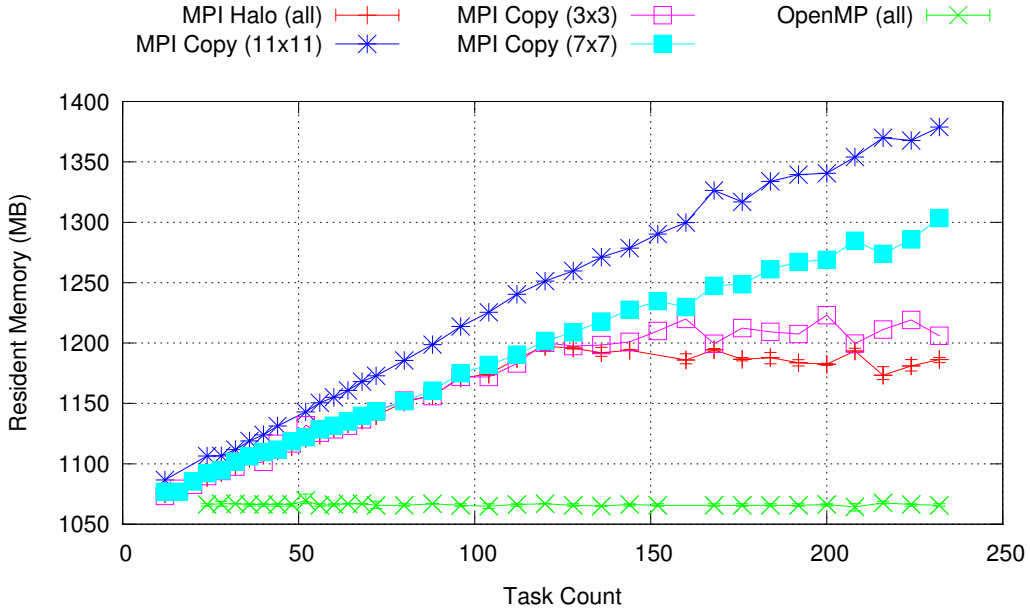
$$f_{n+1}(x,y) = \sum_{i=\frac{-L}{2}}^{\frac{L}{2}} \sum_{j=\frac{-H}{2}}^{\frac{H}{2}} W_{i,j} \times f_n(x+i, y+j) \qquad (3)$$

By applying Equation 3 to an image $f(x,y)$ with a kernel $W_{i,j}$, we are able to model a problem with a controllable level of spatial dependency that is linked to kernel size. Two meshes are required to be able to compute f_{n+1} from f_n. Configuration allowing zero-copy as remote cells are not invalidated during a given time-step, guaranteeing coherency. In order to benchmark the halo-cell method, we attempted to optimize the computation, in particular, completely recovering communications with inner domain computation. We relied on a 1D split to yield the lowest ghost-cell ratio, in order to benchmark our implementation in the less favorable spatial configuration. Our benchmark consists in applying a mean filter twenty times to a three-channel 5616×3744 RGB image stored in double precision. The image is evenly split over processes and ghost cell exchange are done for each channel. We ran the same code in three configurations: (1) with our `MPI_Halo` abstraction, (2) forcing the allocation of halo-cells buffers to behave as a process-based MPI, and (3) on a single MPI task using OpenMP-based parallelism on convolution loops. In each case, we measured computation time and the overall memory usage.

Figure 9(a) presents the total time spent in computing. Note that for readability, we merged `MPI_Halo` curves with those of process-based MPI (copy), as they were very similar (see error bars). This behavior is due to the fact that communications are completely recovered, mitigating their impact on performance. In general, the MPI approach has better performance than the OpenMP one. We believe that the explicit data splitting enforced by MPI limited NUMA effects. Moreover, MPI does not incur OpenMP's fork-join overhead. However, when more that one thread is running on each core, OpenMP is less subject to noise and achieves better performance for certain thread counts. In summary, our `MPI_Halo` approach improved performance when compared to OpenMP with one thread per core (less than 60 tasks), but yielded less stable results with more threads. We believe this is due to the underlying runtime.



(a) Computation time.



(b) Memory Usage (Resident)

Figure 9: Measurement results for our convolution benchmark.

Figure 9(b) displays the amount of resident memory used just after computation. All curves describing `MPI_Halo` were merged, being very close (see error bars) due to the regular splitting. We did the same for OpenMP, as data-sets are identical. It can be seen that even in 1D, there is a non-negligible difference in terms of memory when systematically using halo-cells buffers (process-based approach). The difference increases as expected with the number halo cell layers. However, it seems that the MPI runtime also uses memory for each MPI task (1.05 MB per task, verified with a trivial program), mitigating memory gains when compared to OpenMP. Nonetheless, `MPI_Halo` successfully removed ghost cell buffers growing up to to one third of data-set size, saving a non-negligible amount of memory.

In summary, we shown that our `MPI_Halo` implementation actually provides memory gains on a many-core device when compared to process-based approach. In complement, we also demonstrated a performance gains due to a better locality. The most important result is that our approach succeeds in maintaining the MPI programming model while transposing application to a many-core device.

5. RELATED WORK

Effective memory management on many-core devices is tightly linked to efficiently programming an Exascale machine with billions of threads. In light of memory constraints in an exascale environment, programmers have four different parallelism alternatives[7]: (1) distributed-memory, (2) shared-memory, (3) accelerators and (4) logical address spaces.

Distributed-memory approaches are reaching limits on new many-core hardware as splitting the computation domain inevitably leads to both communication and memory overhead, limiting scalability. The Message Passing Interface is the reference programming model to express distributed-memory computation. Most MPI implementations provide optimized intra-node communications by relying on various approaches[2]: shared memory segments as in MPICH Nemesis[3], kernel modules as in KNEM[10], direct copy for thread-based MPI[8, 15] or even the network interface card. Our approach differs as we are directly accessing data, whereas the message passing standard imposes a copy.

Shared-memory programming (e.g., using OpenMP[5], Cilk[1], Intel TBB, Pthread) is memory-efficient when memory is physically-shared, but must be coupled (hybridized) with distributed-memory support (generally MPI) in order to scale to more than one node. In such context, shared-memory models limit the number of distributed-memory areas, and by extension the number of halo-cells. Accelerators have the advantage of being energy efficient (simpler cores) while providing important computing power through models such as Cuda and OpenCL. Nonetheless, the necessity of transferring the data-set prior to the computation contributes to parallel and memory overhead, requiring careful memory management (e.g., device caches) and latency hiding to harness the full power of such devices.

Models based on logical-memories emulate a shared-memory address space on top of a distributed memory one. Partitioned Global Adress Space (PGAS) exist either as a language (e.g., UPC[11], Co-Array Fortran[14]) or as a library (e.g., OpenSHMEM [4], Global Arrays[13]). Such models provide a way of expressing distributed computation as it would have been done in shared memory, in general by relying on the Remote Memory Access (RMA) capabilities of the network interface card. Regarding memory, some PGAS implementation require a temporary buffer cache to efficiently abstract communications in-between processes[6].

There are several programming models providing both advantages and constraints when it comes to addressing the performance factors of the domain decomposition scenario we are considering for future exascale machines. Clearly, the most suitable way of programming upcoming architectures while transitioning an existing code appears to be model mixing. One challenge such an approach brings is the combination of different runtimes, but it also opens interaction possibilities in-between models. This paper takes advantage of MPC[15] which is an unified MPI+OpenMP runtime. Other initiatives such as MVAPICH2-X [11], HMPI[8], FGMPI[12] also highlight the potential of mixing state of the art models with new approaches in order to unlock new parallelism opportunities.

A work close to our current implementation is the ownership passing interface described by Friedley et al.[9]. Using HMPI, they proposed a compact interface comparable to send and receive but sending only a pointer when an MPI process is accessible in shared memory and relying on a normal copy otherwise. In this paper, we defined an interface to match buffers two by two while hiding buffer allocation complexity. Moreover, by construction ghost cell exchanges involve buffers of the same size. Due to this layout, the allocation is only required when the remote is not accessible in shared memory. Otherwise, the direct pointer is returned, meaning that there are no ghost cells, as only a pointer to the target mesh region is sent. Consequently, our approach cannot be strictly viewed as ownership passing, for example, in Figure 8, *local_left* will always be registered to the same pointer *mesh* (current input mesh). Instead, it provides the remote process with a "view" of current input mesh left boundary. Approach which in complement of avoiding copies (pointer transfers) opens the way for memory saving thanks to buffer aliasing – exchanging buffer views instead of buffers.

6. CONCLUSION

In this paper, we discuss the scaling problems that can be expected when porting applications using domain decomposition to future generation platforms with greater cores counts and reduced memory per core. The memory and communication overheads were modeled by analyzing ghost-cell ratios. Our formula showed that a basic 3D problem is already too large to reach the 1% ratio on current devices. The general outcome of our analysis is what motivates the use of shared-memory parallelism, and what suggests to many application developers to use hybrid programming methods. In contrast, we introduce the `MPI_Halo` abstraction as a convenient way of describing ghost-cell exchanges, using named buffers, triggering all communications at once, and providing error checking. By abstracting halo-cells buffers, this method also exposes zero-copy opportunities when processes are located on the same node. Using the MPC thread-based MPI implementation, we are able to take advantage of `MPI_Halo`, achieving performance gains on a representative benchmark. Not only memory savings are possible when compared to classical process-based splitting, but computation time is better than OpenMP due to the locality enforced by the MPI model. In conclusion, our `MPI_Halo` approach can be used to port pure MPI applications to many-core devices with high performance, while retaining the opportunity for hybrid parallelism improvements.

7. FUTURE WORK

MPI is a parallel programming model known by all HPC developers. Our future work will continue to investigate potential MPI model extensions for shared-memory. Indeed, as the standard allows tasks to be run in threads, we believe that new approaches can be developed to run MPI applications on many-core devices. Our plan is to pursue ideas involving topological communicators and hybrid work-sharing constructs.

8. REFERENCES

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[2] D. Buntinas, G. Mercier, and W. Gropp. Data transfers between processes in an SMP system: Performance study and application to MPI. In *Parallel Processing, 2006. ICPP 2006. International Conference on*. IEEE, 2006.

[3] D. Buntinas, G. Mercier, and W. Gropp. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1. IEEE, 2006.

[4] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith. Introducing OpenSHMEM: SHMEM for the PGAS community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010.

[5] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1), 1998.

[6] B. Dalton, G. Tanase, M. Alvanos, G. Almasi, and E. Tiotto. Memory Management Techniques for Exploiting RDMA in PGAS Languages. In *Workshop on Languages, Compilers and Parallel Computing*, 2014.

[7] J. Dongarra et al. The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 2011.

[8] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine. Hybrid MPI: efficient message passing for multi-core systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013.

[9] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma. Ownership passing: efficient distributed memory programming on multi-core systems. In *ACM SIGPLAN Notices*, volume 48, pages 177–186. ACM, 2013.

[10] B. Goglin and S. Moreaud. KNEM: A generic and scalable kernel-assisted intra-node mpi communication framework. *Journal of Parallel and Distributed Computing*, 73(2), 2013.

[11] J. Jose, M. Luo, S. Sur, and D. K. Panda. Unifying UPC and MPI runtimes: experience with MVAPICH. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*. ACM, 2010.

[12] H. Kamal and A. Wagner. FG-MPI: Fine-grain MPI for multicore and clusters. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, April 2010.

[13] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2), 1996.

[14] R. W. Numrich and J. Reid. Co-Array Fortran for parallel programming. In *ACM Sigplan Fortran Forum*, volume 17. ACM, 1998.

[15] M. Pérache, H. Jourdren, and R. Namyst. MPC: A Unified Parallel Runtime for Clusters of NUMA machines. In *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, Euro-Par '08, Berlin, Heidelberg, 2008. Springer-Verlag.