Hi Dan, and Lukas,

After today's discussions, I am convinced we should be able to integrate Lukas' *OpenCL*-based strategy for accelerator/cpu integration with the PC2 development codelets with minimal effort. This is low-hanging fruit and I believe we should jump on it. If you agree, I'd like to discuss some of the specific details about how each of our codes work and make a plan for integration upon which we can all agree. I am just hoping to start a discussion here, not provide a complete description.

Figures 1, 2 and 3 are cartoons of my understanding of the *OpenCL* code (does it have a proper name?) and *PlasCom2* development codelets. Lukas, please comment and/or correct this if it's off-base. My understanding of the *OpenCL* procedure is as follows:

1. Initialization – the domain is coarsely partitioned, the CPU and GPU regions are "decided", MPI processes **P** and **M** are created to *handle* the CPU and GPU regions, respectively. Process **P** runs a regular C++ program, while the GPU handler process **M** runs an *OpenCL* program. The GPU/CPU and CPU/CPU halo regions are formed, and all partition buffers are created to **include** the halo regions. All initial data is exchanged so that the overall program is ready to begin stepping.

2. Step – Each process **P** conducts the Nth RBSOR step, using the already available remote halo data in its main buffer. OpenMP loop parallelism is used to thread the procedure and distribute to available processing cores on the host platform. At the same time, GPU handler processes **M** use blocking *OpenCL* directives to advance the solution owned by the accelerator devices.

3. Communication collection & send – Each CPU process **P** copies the local halo data (i.e. that required by remote (CPU and GPU) partitions) into MPI send buffers, and sends (non-blocking) to remote processes (including remote processes of type **P** and **M**. Each GPU process copies local halo data into local buffers dedicated to communication with its driver process **M**. The GPU driver process **M** pulls the halo data from the GPU using *OpenCL* directives, and packs the data into MPI communication buffers which it uses to send to remote processes (including the local parent of type **P**, and remote processes of types **P** and **M**.

4. Receive and unpack – All MPI processes of all types post matching receives to all the posted sends, and perform unpacking of the received data into local main buffers. For processes of type **M**, the data is pushed to the device exchange buffer, and then from the device exchange buffer into the device partition's main buffer using blocking *OpenCL* directives.

5. Update N - Go to Step 2

I propose that we do a few of things:

Update Lukas' OpenCL code – Port the advection kernels from the PC2 development codelets and RK4 to the OpenCL integrator, and conduct the verification test on the results. Currently, we only really know that the OpenCL integrator doesn't fail to run. We have a one-file, flat procedural implementation of the 1dadvection that should be quite easy to port to Lukas' framework. This will also give us an idea of what it takes to implement the time-marching driver in the *OpenCL* program run by MPI processes of type **M**.

Update Codelet 4 to use OpenCL – The general plan would be to take the interior part of the partition interval, I, and dispatch it to the GPU through using Lukas' mechanism of spawning off an MPI process of type **M** to manage the GPU through *OpenCL*. The CPU part of the partition interval would likely retain only the local halo regions, B_i, and domain boundaries. The fact that some remote processes are of type **M** and handle GPU devices should be transparent to the current operation of Codelet 4.

There are a few additional things we should discuss about using asynchronous communication and asynchronous *OpenCL* to overlap communciation and computation for further gains in Lukas' code, as well as to jibe with the current Codelet4 control flow. Lukas and I have reviewed a potential control flow, and identified a couple of minor related buffering issues. More details on this can be provided as our discussions continue (if we agree they should continue).

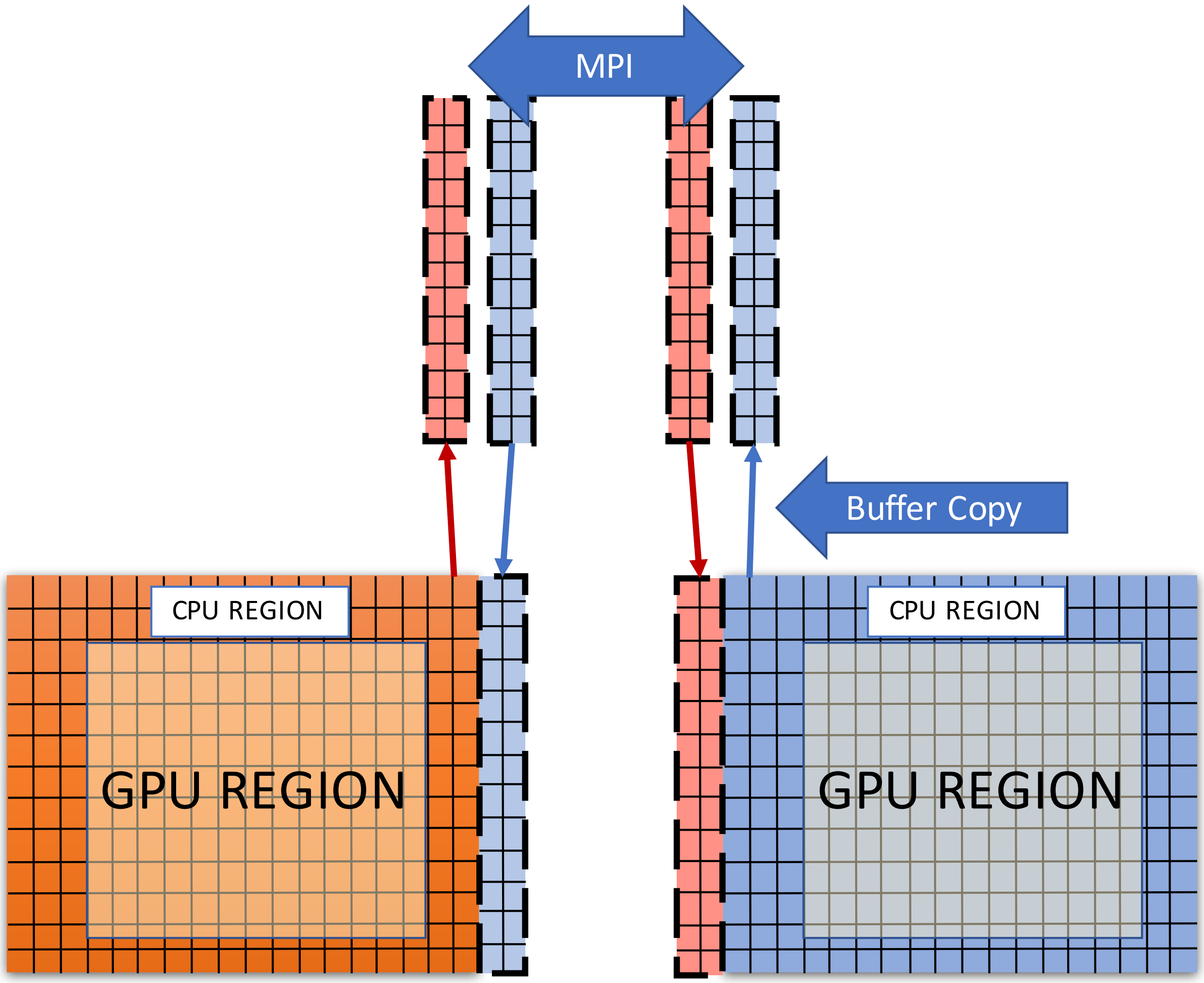

Figure 1: Partition arrangements and communication buffers for OpenCL integrator. An initial *coarse* partitioning into some number of partitions is followed by a finer partitioning of each coarse partition where some subset of the partition is sent to a GPU. In general, the GPU region may collide with the coarse partition boundary, but in practice it may be better if it does not. Each MPI process that handles some portion of the domain (i.e. some partition), communicates data in the halo regions by performing copies to-and-from communication buffers that are used in MPI exchanges. The grid and state buffers are allocated such that they *include* the remote halo regions (i.e. the regions that will contain data owned by remote partitions).

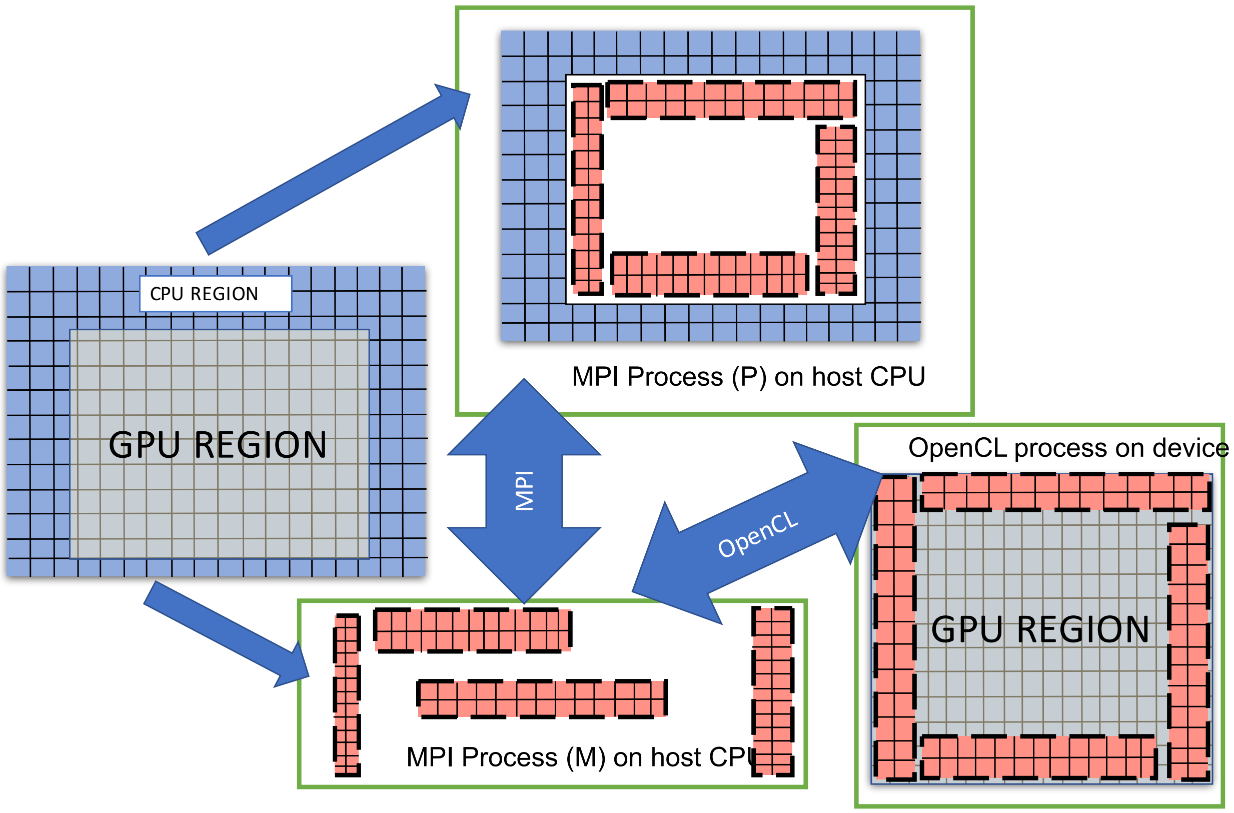

Figure 2: Details of single coarse partition of the OpenCL integrator – *work* partitioning. The CPU region(s) (possibly a disparate collection of them) are handled by MPI process **P** on the compute host, and the GPU region (a contiguous subset of the coarse partition) is handled by MPI process **M** on the compute host. MPI process **M** is simply a *handler* for *OpenCL* program that runs on the device. The handler process owns the MPI communication buffers used in communicating the gpu/cpu halos between the two host-local MPI processes, and manages the control flow of the *OpenCL* program. If the GPU region actually collides with the coarse partition boundaries, then the handler process is capable of sending data directly to the corresponding MPI process that requires the halo data. GPU/CPU communication and control flow handling between the *OpenCL* program and the MPI process **M** are acutated by *OpenCL* directives, and involves the same copy,transfer,copy sequence used in the MPI-MPI exchanges.

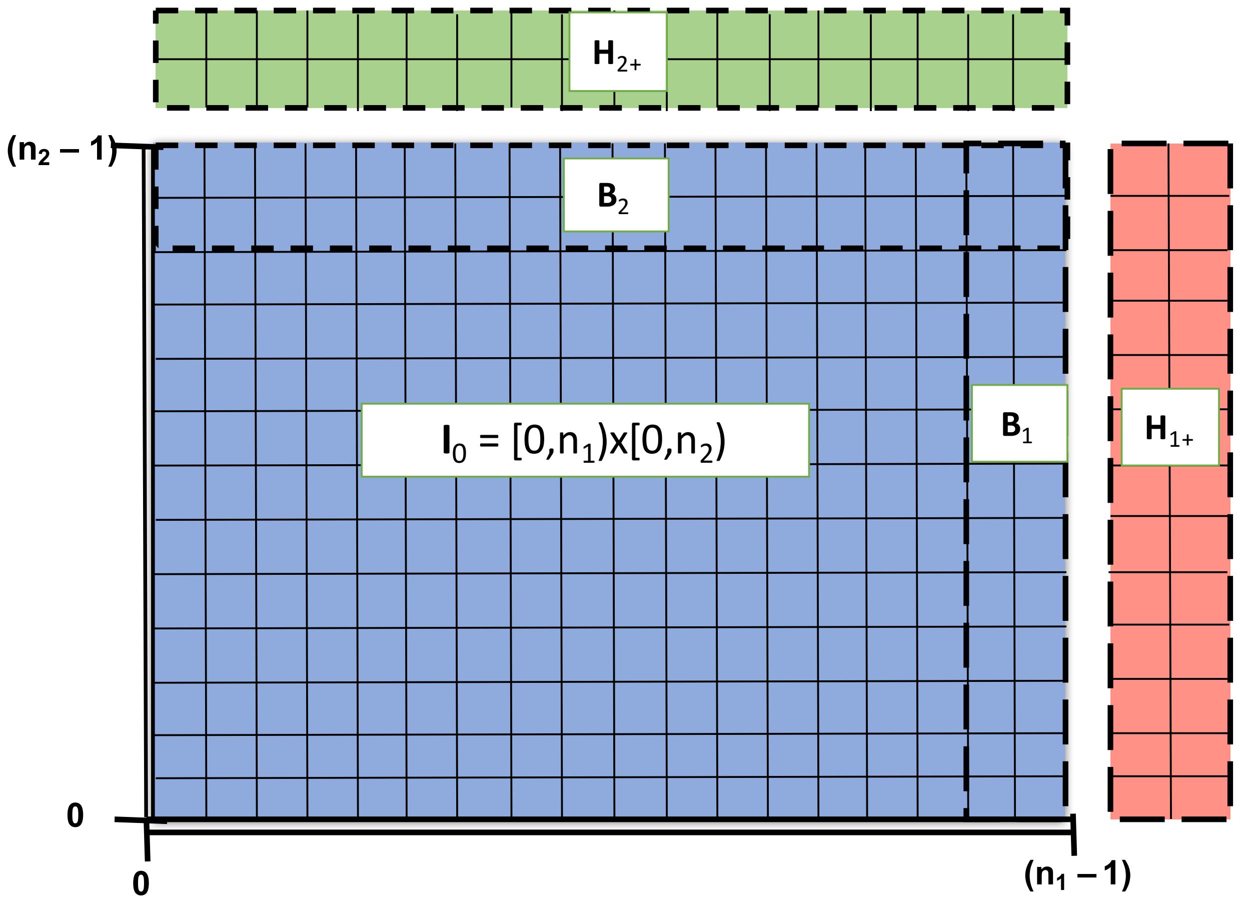

Figure 3: PlasCom2 development codelet partition arrangements, regions, and communication regions. In Codelets 3 & 4, multiple levels of partitioning result in a set of partition intervals, I_p. The grid and state buffers are allocated such that the *remote halo regions* specified by intervals $H_{i\pm}$ are not included. The *local halo regions* specified by intervals $B_{i\pm}$ are the regions in the local partition that are required by remote partitions. For each region that requires communication with remote partitions, matching send and receive buffers are created.