

JacSolver Example

This OpenCL example application illustrates how to use OpenCL together with OpenMPI, an Open Source implementation of MPI.

What is MPI?

MPI is a language-independent communications protocol used to program parallel computers. Both point-to-point and collective communication are supported. MPI "is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation." The goals of MPI are high performance, scalability, and portability. MPI provides APIs for:

- Topology querying and creation of neighbors in a Cartesian layout
- Synchronous and asynchronous sends and receives including simultaneous send/receive
- Broadcast
- Scatter and gather functions
- Reduce operations (e.g. MPI_SUM, MPI_MIN and MPI_MAX)
- Parallel I/O (MPI-2)

One advantage of using MPI is that the same code can be used either across a cluster of multiple machines or within one machine that has multiple cores.

The Compute Problem to be Solved

This example solves the Laplace steady-state heat equation in two dimensions:

$$\nabla^2 \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

This equation is one of the simplest examples of an elliptic partial differential equation (PDE). The following Dirichlet boundary conditions are used:

$$\begin{aligned}\phi(x,0) &= \sin(\pi x) & 0 \leq x \leq 1 \\ \phi(x,1) &= \sin(\pi x)e^{-\pi} & 0 \leq x \leq 1 \\ \phi(0,y) &= 0 & 0 \leq y \leq 1 \\ \phi(1,y) &= 0 & 0 \leq y \leq 1\end{aligned}$$

In terms of the heat equation, one physical interpretation of this problem is fixing the temperature at the four edges of the unit square and allowing heat to flow until a steady state is reached. The temperature distribution in the interior of the plate (represented by a real number in the range [0,1]) corresponds to the solution to the Dirichlet boundary problem. A visual representation of the steady state condition is shown in the picture on the next page where the range of values [0,1] is given by a color value from blue to red. The example code outputs a bitmap file in PPM format.

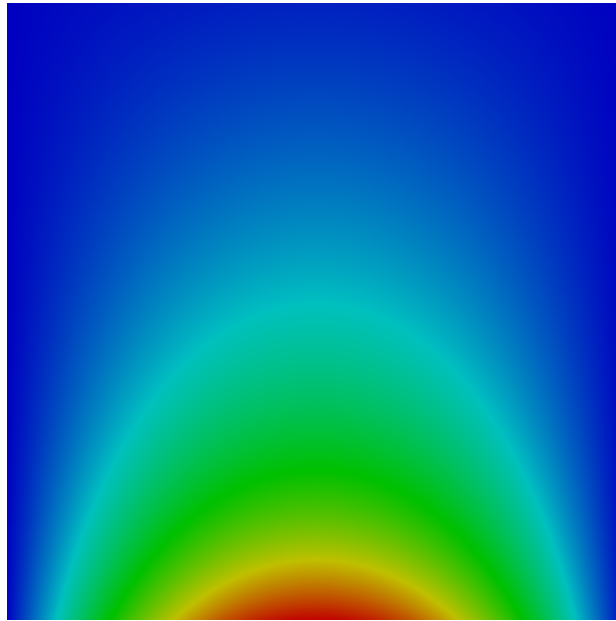
There are several standard methods to solve the Laplace equation with Dirichlet boundary conditions. An analytical method can be used to separate this PDE into two independent ordinary differential equations (ODEs) in x and y . Solving these ODEs separately and substituting in the boundary conditions results in the following analytical solution:

$$\phi(x,y) = \sin(\pi x) e^{-\pi y} \quad 0 \leq x \leq 1, 0 \leq y \leq 1$$

Another way to solve the problem is to use a forward finite difference method which approximates the second derivatives in x and y using a difference equation and linearly discretizing the values in each dimension:

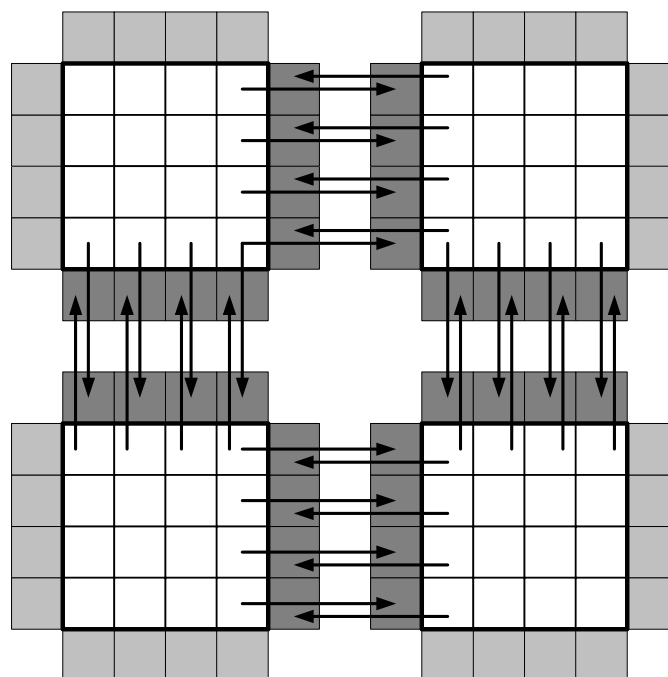
$$\begin{aligned}\phi_{i,j} &= \frac{1}{4} (\phi_{i-1,j} + \phi_{i+1,j} + \phi_{i,j-1} + \phi_{i,j+1}) \\ \text{where } i &= 0,1,2 \dots m, \quad j = 0,1,2 \dots n \\ \text{and } \Delta x &= \frac{1}{m}, \quad \Delta y = \frac{1}{n}\end{aligned}$$

JacSolver Example



The difference equation leads to the result that any point in the array is the average of the four surrounding points in a first order 5-point stencil. A Jacobi iterative solver is used because it is simple to understand and is N-parallel – meaning that the new array values can be calculated all at once, in parallel. This is ideal for OpenCL accelerators and also clustering using MPI. The Jacobi iteration works on an array of dimensions $m \times n$ that is seeded with the boundary condition values and an initial guess of 0.5 – this is $\phi^{(0)}$. The new values for array elements that are not on the boundary are calculated based on (7) – this is $\phi^{(1)}$. Then $\phi^{(2)}$ is calculated and so on until the differences of every array element between two successive iterations is less than a tolerance value of 0.00001.

The standard technique of ghost (halo) cells is used to facilitate parallel computation when Jacobi solver is part of a MPI cluster. When the iteration has finished, the ghost cells are exchanged with neighboring nodes in the cluster as shown in the diagram below. Non-contiguous ghost cells are copied in and out of intermediate contiguous buffers to reduce the MPI communication overhead.



MPI Details

For the purposes of demonstrating the use of MPI, the number of compute processors can be specified in both the x and y dimensions which allows for horizontal strips, vertical strips, and rectangles (patches). This section describes the changes that were made to the example to support more than one MPI node. Note that if only one MPI node is specified by setting the -p and -q options to the default value of 1, then the code does use MPI.

The *ranks* global variable together with the RANK_INDEX defines are used to store the rank of the current node plus its 4 neighbors in the each of the 4 directions. A negative number is used to indicate when the current rank doesn't have a neighbor in the given direction.

The *MSG()*, and *ERR()* macros incorporate the MPI rank number so that messages from a particular node can be distinguished from other nodes.

The *init_mpi()* function is used to initialize MPI and setup the MPI Cartesian coordinates so that each node knows the MPI rank numbers for its neighbors in each of the 4 directions (up, down, left and right). The neighbors are used later to exchange ghost data. The *exit_app()* function is used to close down MPI at the end of the application.

The *print_array()* function is used to dump out the contents of the array for small data sizes. MPI synchronization barriers and rank numbers are used so only one output array is printed in the correct order for each MPI rank.

The *ocl_get_device_id()* function is used to allocate an OpenCL device to each MPI rank. If multiple MPI ranks are run on the same machine, then an individual device is needed for each rank.

The *exchange_ghost_cells()* function is used to synchronously exchange ghost cells with each neighbor. The MPI_Sendrecv function is used so that a simultaneous send and receive can be done. Exchanges across the vertical sides are simple because the data is contiguous in the Y direction. Exchanges across horizontal boundaries are a little more complex because the non-contiguous boundary data needs to be copied in and out of buffers that are contiguous. The reason for using a single buffer for gather/scatter is that this results in a single call to MPI_Sendrecv rather than many calls, one for each vertical, non-contiguous ghost cell.

The *read_ghost_cells_from_device()* and *write_ghost_cells_from_device()* functions are used to copy ghost cell data in and out of device memory which is required for MPI data exchanges. The neighbor rank information is used to determine which ghost cell data needs to be copied.

The *ocl_jacobi()* function is the main Jacobi iteration routine. It uses the ghost cell routines listed above to handle MPI changes of ghost cell data but only if there is more than one MPI node. The final operation is a MPI reduce (MPI_MAX) which is used to determine the maximum error across all the MPI nodes. The *reference_jacobi()* function for the reference (non-OpenCL) implementation also only uses the ghost cell exchange and MPI reduce operations when there is more than one MPI node.

The *set_boundary_conditions()* function sets the appropriate boundary conditions for each MPI node using the neighbor rank information.

The *save_bitmap()* function creates a RGB bitmap in PPM format for each rank; it does not generate a single bitmap across all of the ranks. The bitmap file for each rank (except ghost cells) is uniquely named file to reflect the X and Y rank number. The user will need to join the bitmaps together into one larger one, if required.