NA VEC SCATTERS IN PETSC

Joseph Pusztay

Based on: https://arxiv.org/pdf/1612.08060.pdf

University at Buffalo The State University of New York

Overview

- What is PETSc?
- What is multigrid, and why does it matter?
- Communication in PETSc, and examples
- The Node Aware Algorithm
- Implementation Changes
- Preliminary Results
- Future Plans for this Project

Reference Work: https://arxiv.org/pdf/1612.08060.pdf

The Portable Extensible Toolkit for Scientific Computing

- A library offering a collection of APIs and data structures for scientific computation
- Includes non-linear solvers, finite elements, a suite of preconditioners, particle methods, time steppers, etc.
- Removes low level concerns from the user
- A plethora on usage examples for the various operations

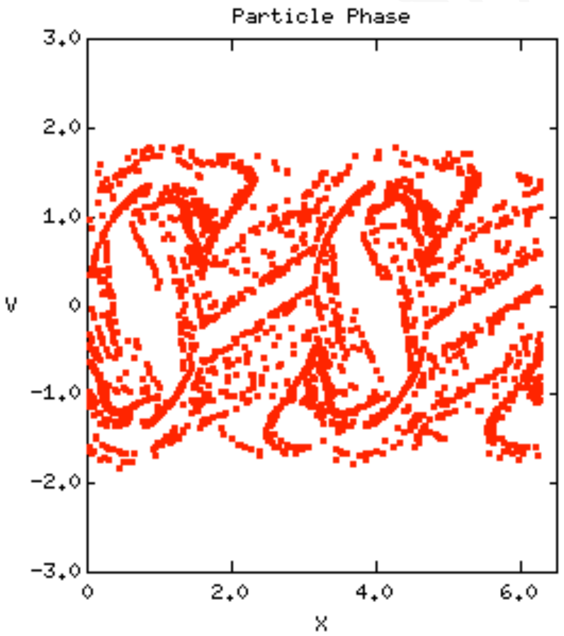


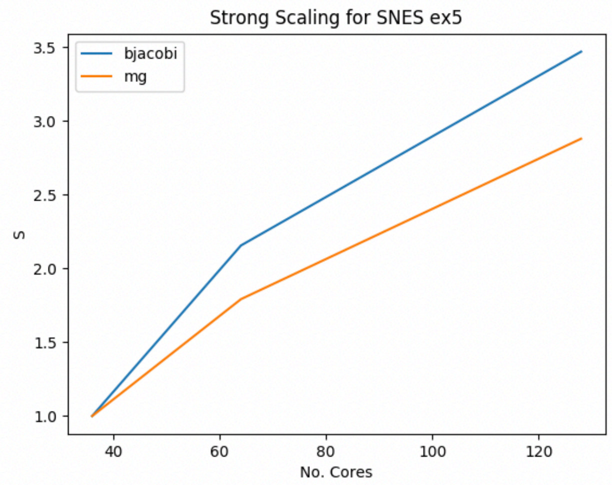
Figure 1: Vlasov-Poisson Particle in Cell Simulation for the Two-Stream Instability problem

Benefits of PETSc

- Abstracted API layers remove parallelization concerns from the user
- Mature code base
- Actively developed and maintained
- Wide spread utilization
- Open Source
- Cross platform

High Level MG (an extremely brief introduction)

- We would like to solve some PDE on a grid, but:
- Grids that are too coarse may lose information about the problem, resulting in extraneous errors
- Refining the grid too much becomes extremely expensive computationally
- Multigrid handles both of these issues by handling the grid at various levels of coarseness
    - The goal is to catch long wave and short wave errors (High/ Low frequency waves in the solution)
- This involves many communications between segments of the grid



Strong scaling for PETSc SNES EX5 using 2 Skylake nodes. This example models solid fuel ignition in 2D. Observed is the strong scaling of multigrid and block jacobi preconditioners

PETSc Vec Objects

- Vec objects represent vectors and have mathematically relevant operations defined
- Parallel Vec objects are able to be shared between nodes
    - Vec Scatter/Gather operations depend on global indexing over a compatible communicator
    - Local to global index mapping is maintained to perform parallel operations (Scatter)

NAPSpMV

- Sparse Matrix-Vector operations are found in various situations (for example, MG)
- Node to node communications become heavy as processors attempt to share information across nodes, resulting in large communication overheads
- Reduce overhead by relying on the configuration of processors within the nodes
  - Change the communication from many to many (processors), to one (processor) to many (nodes) and many (nodes) to one (processor), then unpack communication within a node

Figure from: https://arxiv.org/pdf/1612.08060.pdf

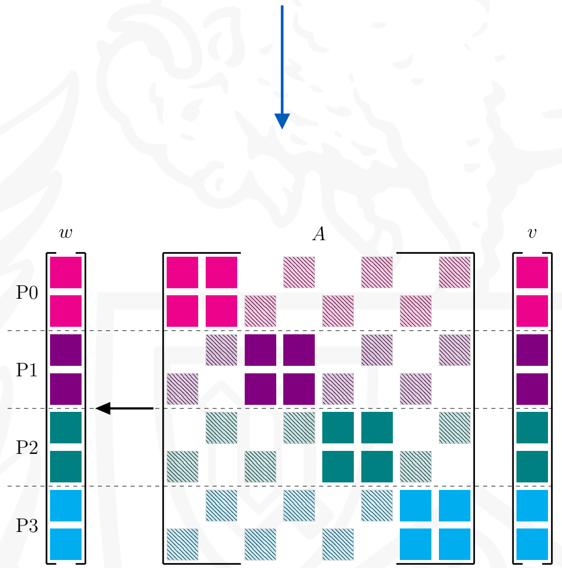


Figure 1: A matrix partitioned across four processes, where each process stores two rows of the matrix, and the equivalent rows of each vector. The on-process block of each matrix partition is represented by solid squares, while the off-process block is represented by patterned entries.

NAPSpMV Algorithm

Algorithm 2: `local_comm`

Input:
- (p, n) : tuple describing local rank and node of process
- $v|_{\mathcal{R}((p,n))}$: rows of input vector v local to process (p, n)
- `locality`: locality of input and output data

Output:
- ℓ_{recv}: values that rank (p, n) receives from other processes

```
// Initialize sends
for (s, n) ∈ L((p, n), locality) do
    for i ∈ J((p, n), (s, n), locality) do
        ℓ_send ← v|_R((p,n))_i
    MPI_Isend(ℓ_send, ..., (s, n), ...)

// Initialize receives
ℓ_recv ← ∅
for (s, n) s.t. (p, n) ∈ L((s, n), locality) do
    MPI_Irecv(ℓ_recv, ..., (s, n), ...)

// Complete sends and receives
MPI_Waitall
```

*

Algorithm 3: `NAPSpMV`

Input:
- (p, n): tuple describing local rank and node of process
- $A|R$: rows of matrix A local to process (p, n)
- $v|R$: rows of input vector v local to process (p, n)

Output:
- $w|\mathcal{R}$: rows of output vector $w \leftarrow Av$, local to process (p, n)

```
A_on_process = on_process(A|R)
A_on_node = on_node(A|R)
A_off_node = off_node(A|R)

b_ℓ→ℓ ← local_comm((p, n), v|R, (on_node → on_node))
b_ℓ→nℓ ← local_comm((p, n), v|R, (on_node → off_node))

// Initialize sends
for (q, m) ∈ G((p, n)) do
    for i ∈ I((p, n), (q, m)) do
        g_send ← b^i_ℓ→nℓ
    MPI_Isend(g_send, ..., (q, m), ...)

// Initialize receives
g_recv ← ∅
for (q, m) s.t. (p, n) ∈ G((q, m)) do
    MPI_Irecv(g_recv, ..., (q, m), ...)

// Serial SpMV for local values
local_spmv(A_on_process, v|R)

// Serial SpMv for on-node values
local_spmv(A_on_node, b_ℓ→ℓ)

// Complete sends and receives
MPI_Waitall

b_nℓ→ℓ ← local_comm((p, n), v|R, (off_node → on_node))

// Serial SpMV for off-node values
local_spmv(A_off_node, b_nℓ→ℓ)
```

*

[* https://arxiv.org/pdf/1612.08060.pdf](https://arxiv.org/pdf/1612.08060.pdf)

Modifications to the Algorithm

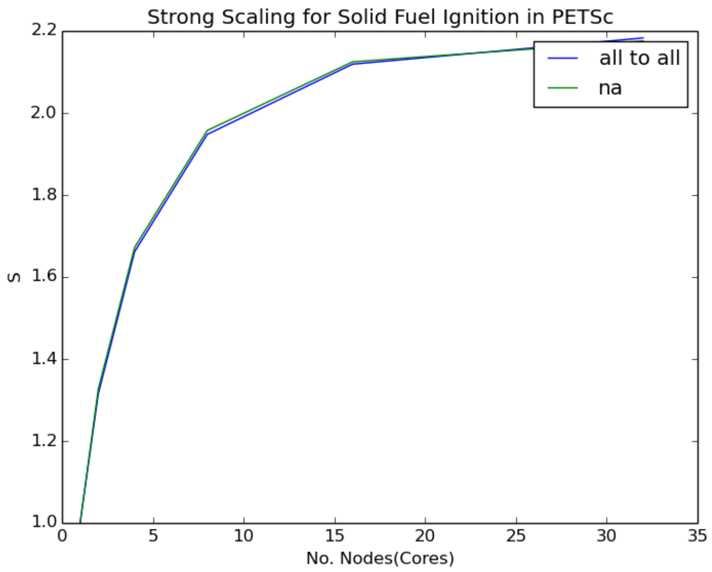
- Petsc global index ordering must be consistent and maintained
    - Separate communicator splits would require additional layers of translation, a global ordering must still be maintained
- Hard code node mapping based on specific run time environment for simplicity
    - Decrease set up time
    - Unnecessary for larger problems (We are running on a small problem)
- Introduced a new VecScatter type to PETSc (~+5000 lines of code)
    - PETSc block size dependent definitions for packing and unpacking sends and receive buffers
    - On creation of a VecScatter context, maintain global ordering but perform MPI_Isend and MPI_Ireceive translations to pack buffers for message passing and routing
- Will this get us our performance gains?

Steps

- Configure a context for VecScatter
  - Compute expected number of messages passed between processors on the node, create sends/receives
  - Compute expected number of messages passed in a buffer between a node, create sends/receives
    - Configure message Packing for internode/intranode communication
- VecScatter
  - Send messages on the node, off node communication goes to a process to pack buffer
  - Node receives message from off node process, unpacks buffer and distributes between the processes
  - Vector Operations are performed and Vector update by backwards scatter.

Scaling: 1PPN

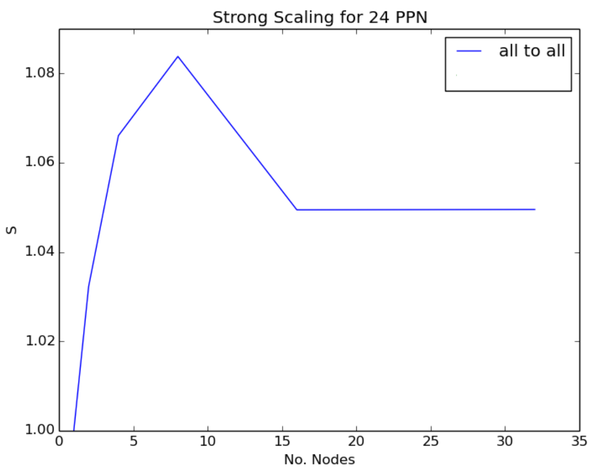
- Build petsc environment with Intel 2019 compilers and intel MPI (versions 2019.5)
- Build PETSc SNES ex5
- Execute on 1-32 nodes
- Expected results for this configuration?
  - Yes



Scatter type	1 nodes	2 nodes	4 nodes	8 nodes	16	32
Standard	1.027E+02	7.8112E+01	6.1827E+01	5.2742E+01	4.8480E+01	4.7059E+01
NA	1.0332E+02	7.877E+01	6.1787E+01	5.2786E+01	4.8628E+01	4.7490E+01

Conclusion and future work:

- What do we expect from Node Aware on this problem?
  - Comparable performance
    - Runs are short and may not fully resolve the effects
  - Does this problem entail enough communication overhead to test the algorithm?
- Future Plans:
  - Send/Receive configurations need some more debugging for more extensible configurations to support more/arbitrary processes per node, and have been rethought since their initial implementation regardless
    - Final library will configure VecScatter contexts from a config file that contains cluster topology for best performance (current thoughts)
  - Increase portability from system to system (aforementioned config files)
  - Plenty of further testing/tweaking! (longer problems, larger problems, more communicationally expensive problems, etc.)
  - And finally, merge request into PETSc



Strong scaling up to 32 nodes with 24 processors per node with multigrid preconditioner. *NA Vec Scatter not pictured

QUESTIONS