MVAPICH
MPI, PGAS and Hybrid MPI+PGAS Library

Machine-agnostic and Communication-aware Designs for MPI on Emerging Architectures

Jahanzeb Maqbool Hashmi, Shulei Xu, Bharath Ramesh, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K. (DK) Panda

{hashmi.29, xu.2452, ramesh.113, bayatpour.1, subramoni.1, panda.2}@osu.edu

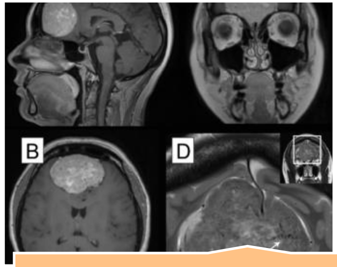
IPDPS'20

Department of Computer Science and Engineering

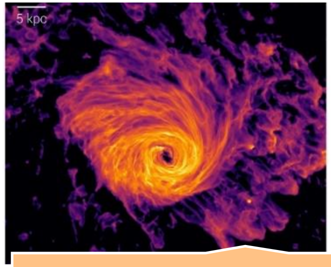
The Ohio State University, Columbus, USA

Diversity in HPC Applications and Architectures

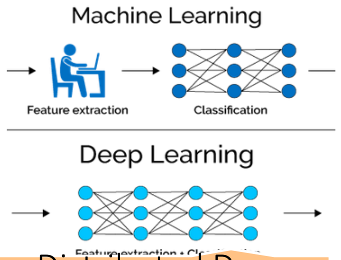
Genome Sequencing



Medical Imaging



Galaxy Formation



Distributed Deep Learning

Multi/ Many-core Processors

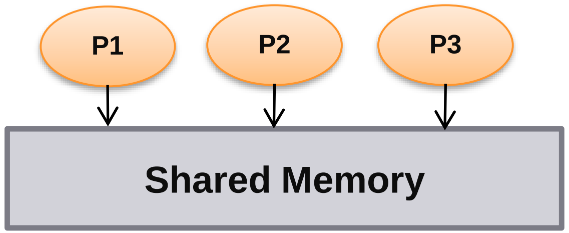
High Performance Interconnects – InfiniBand, Omni-Path
<1usec latency, 100Gbps Bandwidth>

Accelerators / Coprocessors
high compute density, high performance/watt

SSD, NVMe-SSD, NVRAM

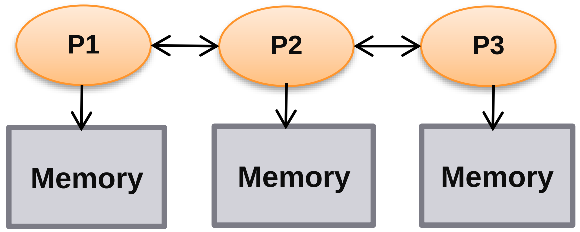
- Multi-core/many-core technologies
- High Performance Interconnects
- High Performance Storage and Compute devices
- Variety of programming models (MPI, PGAS, MPI+X)

Parallel Programming Models



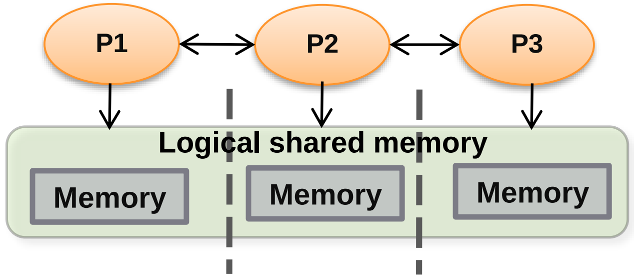
Shared Memory Model

SHMEM, DSM



Distributed Memory Model

MPI (Message Passing Interface)



Partitioned Global Address Space (PGAS)

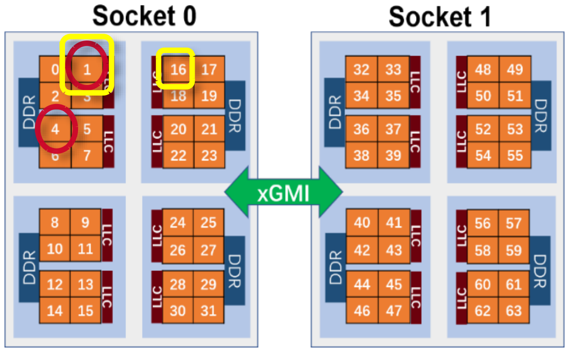
OpenSHMEM, UPC, UPC++, CAF …

- Programming models provide abstract machine models
- Models can be mapped on different types of systems
  - e.g. Distributed Shared Memory (DSM), MPI within a node, etc.
- MPI is the de-facto programming model for writing parallel applications
- MPI offers various communication primitives and data layouts
  - Point-to-point, Collectives, Remote Memory Access
  - Derived Datatypes

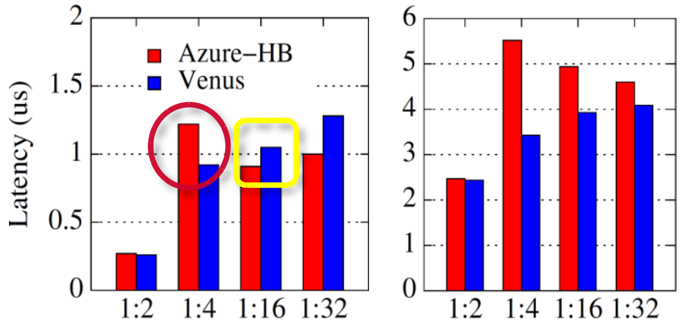
Motivating Examples

- Experiment
  - Run osu_latency benchmark on two AMD EPYC systems (e.g., Azure-HB VM, and native)
- Three observations
  - MPI mappings perform differently on native vs. VM systems
  - The mapping that works for one application, need not work well for another application
  - Even on same architecture, different mappings exhibit different performance
- There is a need for adaptive MPI runtime

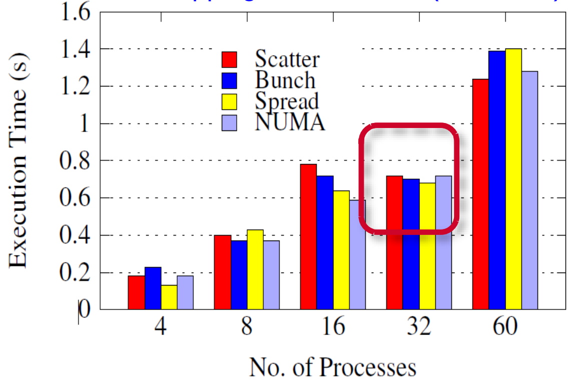
AMD EPYC 7551 Architecture



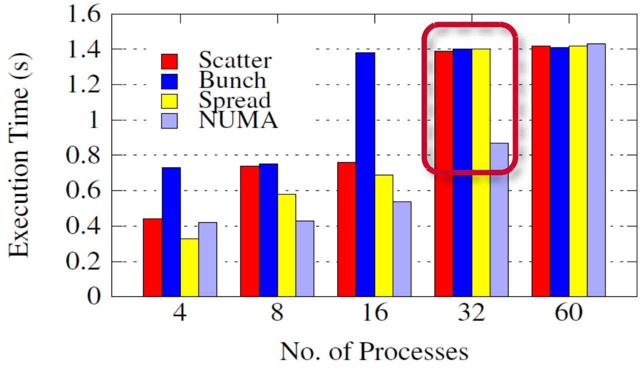
Latency test with various Rank-to-core placements with small (left) and large (right) messages



MPI Mappings on MiniAMR (Azure-HB)



MPI Mappings on MiniGhost (Azure-HB)



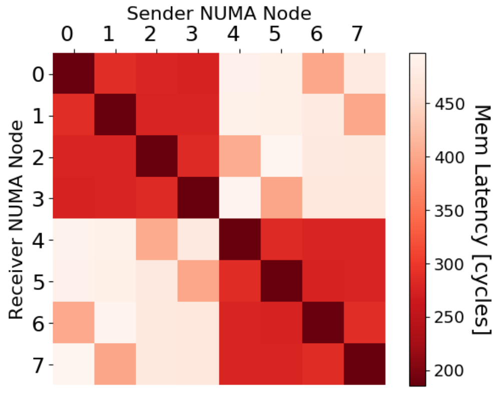
Research Challenges

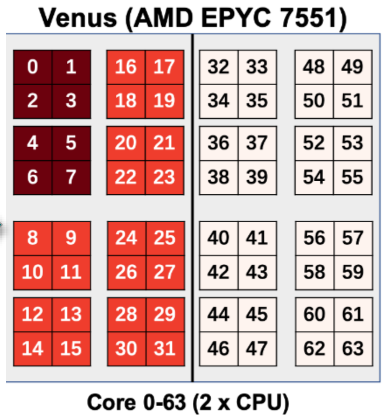
- Challenges due to diverse architectures
  - Multi-cores, Many-cores with SMT
  - Deep memory hierarchies (e.g., NUMA levels)
  - HPC virtual machines configurations (e.g., Azure)
- Challenges due to dynamic application communication patterns
  - Irregular (Graphs) vs. Regular (Near-neighbor, Halo-exchange)
- Research Questions
  - Can we design adaptive MPI process mappings that work with any communication patterns and underlying hardware?
  - How can we design architecture and communication-pattern aware MPI runtime without requiring application changes?
- Proposed Solution
  - Architecture-agnostic and machine-aware adaptive MPI runtime
  - Construct hardware topologies and application communication graphs
  - Design efficient and adaptive MPI rank-to-core mappings

Proposed Design for Constructing Machine Topologies

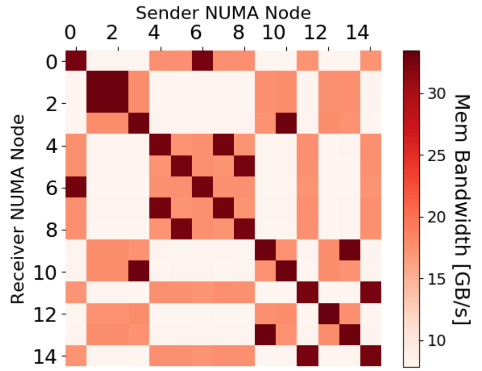
- Problem with state-of-the-art
  - Topology detection rely on static approaches e.g., `hwloc`
  - Do not work well with complex architecture e.g., Azure VM
- Proposed approach:
  - Use online measurements to construct physical to virtual resource graph
    - Use `MCTOP`[2] for low-level (e.g., cache-line level) benchmarking
    - Works regardless of the native and VM systems

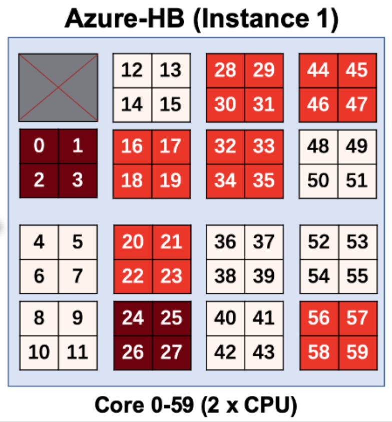
Venus NUMA-NUMA Bandwidth





Azure-HB NUMA-NUMA Bandwidth

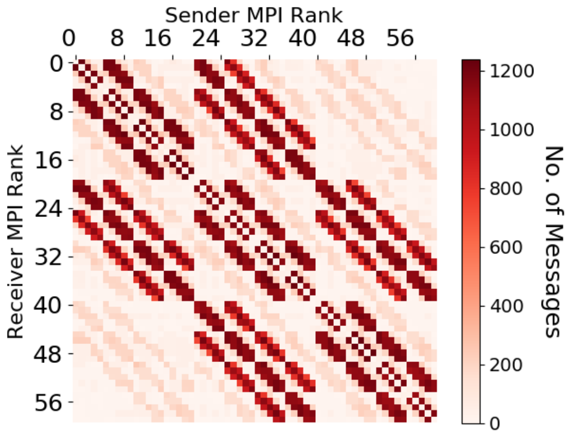




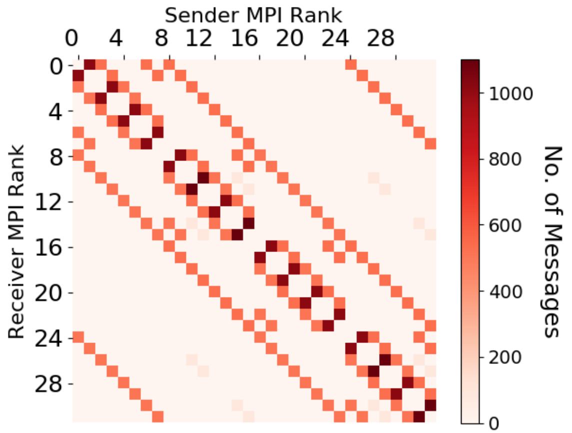
[2] "Abstracting Multi-Core Topologies with MCTOP", Georgios et al. EuroSys'17

Proposed Efficient MPI Rank to Core Mappings

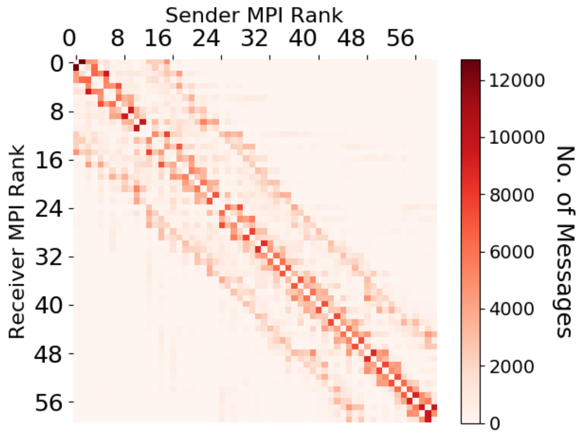
Communication Pattern (AMG)



Communication Pattern (NAS_MG)



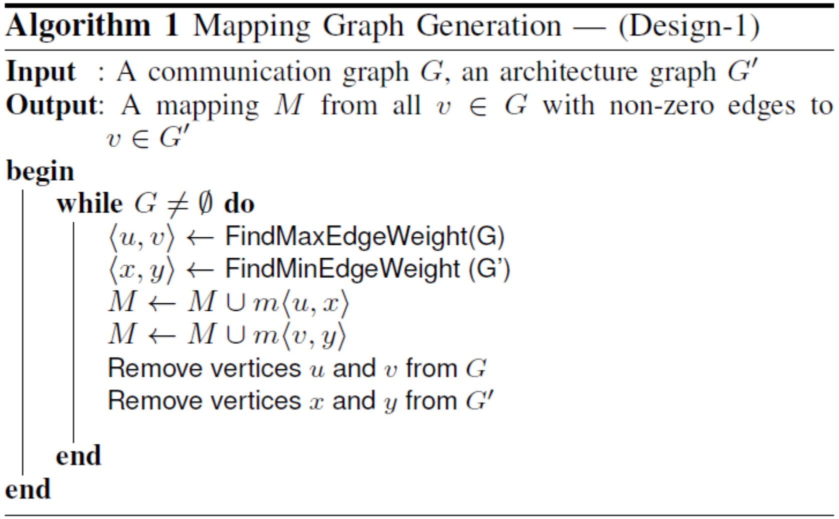
Communication Pattern (MiniAMR)



- **Efficient virtual to physical resource mapping algorithm**
  - Step 1: Construct topology graph (G) using MCTOP
  - Step 2: Adaptively generate communication graph (G’) using MPI_T interception (on-the-fly)
  - Step 3: Greedy style algorithms to map high-cost edges in G’ on to low-latency/high-bandwidth edges in G

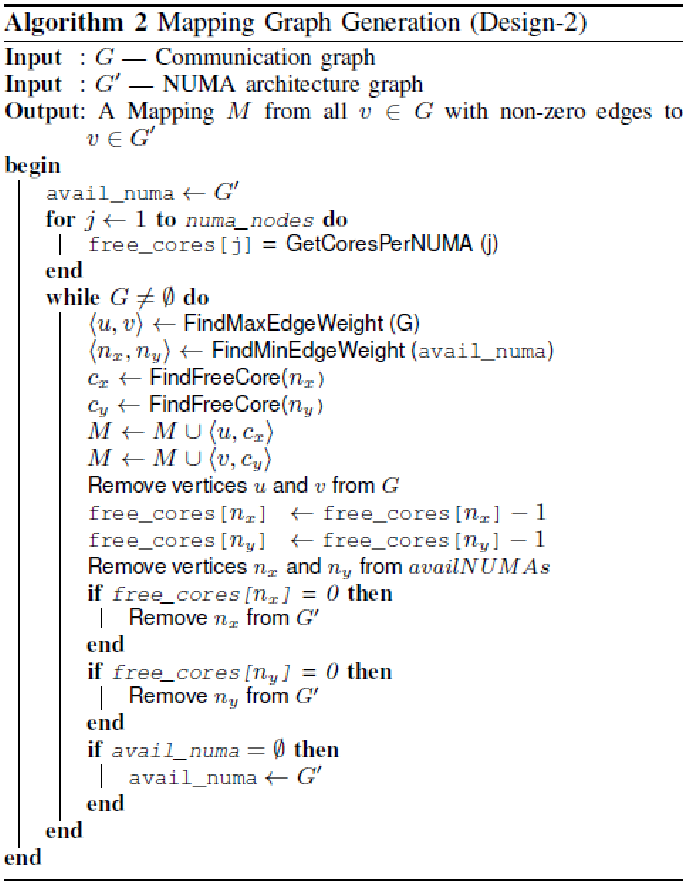
Proposed MPI Rank Mapping Algorithms

- Basic Idea
  - Find highest communication edges in the application graph (G)
  - Find lowest latency edges in hardware topology graph (G’)
  - Construct a list of mappings from G to G’



Algorithm 1 Mapping Graph Generation — (Design-1)

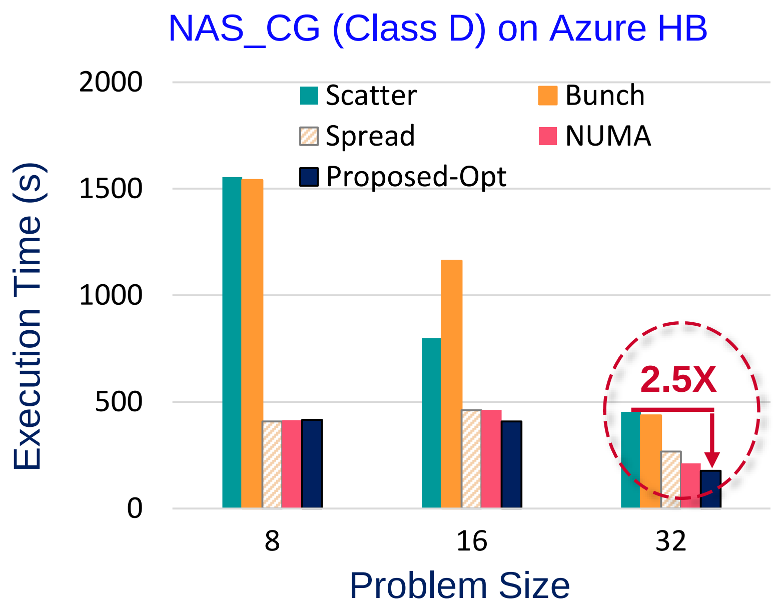
```
Input  : A communication graph $G$, an architecture graph $G'$
Output: A mapping $M$ from all $v \in G$ with non-zero edges to
        $v \in G'$
begin
    while $G \neq \emptyset$ do
        $\langle u, v \rangle \leftarrow$ FindMaxEdgeWeight(G)
        $\langle x, y \rangle \leftarrow$ FindMinEdgeWeight (G’)
        $M \leftarrow M \cup m\langle u, x \rangle$
        $M \leftarrow M \cup m\langle v, y \rangle$
        Remove vertices $u$ and $v$ from $G$
        Remove vertices $x$ and $y$ from $G'$
    end
end
```

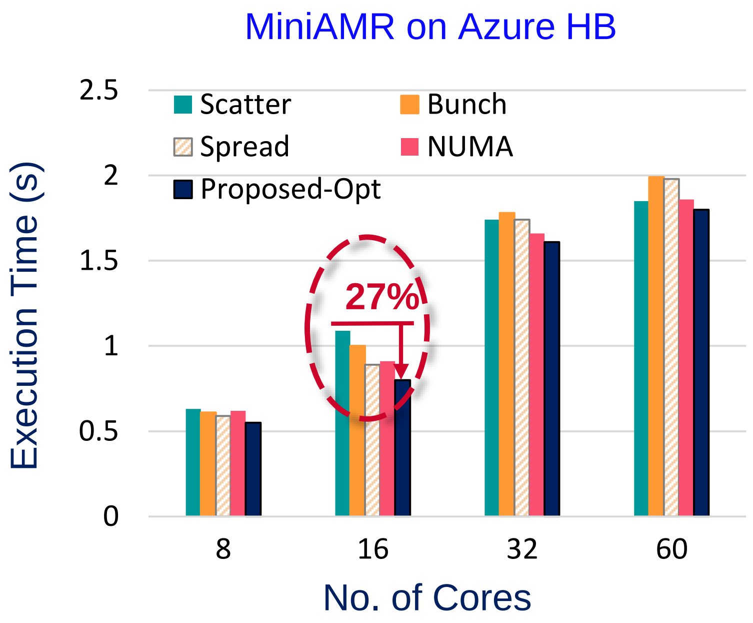


Algorithm 2 Mapping Graph Generation (Design-2)

```
Input  : $G$ — Communication graph
Input  : $G'$ — NUMA architecture graph
Output: A Mapping $M$ from all $v \in G$ with non-zero edges to
        $v \in G'$
begin
    avail_numa $\leftarrow G'$
    for $j \leftarrow 1$ to numa_nodes do
        free_cores[j] = GetCoresPerNUMA (j)
    end
    while $G \neq \emptyset$ do
        $\langle u, v \rangle \leftarrow$ FindMaxEdgeWeight (G)
        $\langle n_x, n_y \rangle \leftarrow$ FindMinEdgeWeight (avail_numa)
        $c_x \leftarrow$ FindFreeCore($n_x$)
        $c_y \leftarrow$ FindFreeCore($n_y$)
        $M \leftarrow M \cup \langle u, c_x \rangle$
        $M \leftarrow M \cup \langle v, c_y \rangle$
        Remove vertices $u$ and $v$ from $G$
        free_cores[$n_x$] $\leftarrow$ free_cores[$n_x$] $- 1$
        free_cores[$n_y$] $\leftarrow$ free_cores[$n_y$] $- 1$
        Remove vertices $n_x$ and $n_y$ from $availNUMAs$
        if free_cores[$n_x$] = 0 then
            Remove $n_x$ from $G'$
        end
        if free_cores[$n_y$] = 0 then
            Remove $n_y$ from $G'$
        end
        if avail_numa $= \emptyset$ then
            avail_numa $\leftarrow G'$
        end
    end
end
```

Application-level Benefits of the Proposed Designs





- Up to 27% improvement for MiniAMR and 2.5X improvement for NAS_CG kernel when compared with static MPI mapping policies in MVAPICH2
- For more detailed results, please refer to the paper

Conclusion

- Modern multi-/many-core architecture and applications are becoming more diverse
  - HPC over cloud systems are adding more complexity
- Application communication patterns are dynamic
  - Irregular vs. regular
- Existing MPI runtimes are rigid when it comes to mapping application threads to hardware resources
  - Mapping policies use static tools
  - Unaware of the application’s communication and underlying hardware topologies
- Proposed novel hardware-agnostic and communication-aware MPI runtime to efficiently map MPI ranks on to hardware resources transparent to the application
- Significant performance benefits on real HPC system and application kernels
- In future, we plan to extend these ideas to accelerators e.g., GPUs and large-scale HPC clusters