ZedWulf - A Zynq SoC cluster for Energy-Efficient Acceleration of Graph Problems

Abstract—
In recent times, we see an increasing amount of research interest in exploring the usage of ARM architectures in High Performance Computing (HPC). While pure ARM chips have historically been lacking the performance edge over x86, ARM-FPGA hybrid designs such as the Zynq SoC can be the potential catalysts. We run micro-benchmarks to understand Memory System, AXI-Accelerator Coherency Port (ACP) and Network characteristics of a Zynq-based platform (Zedboard) in the presence of Xillinux OS [22]. We build a cluster composed of 32 such Zynq-based devices (ZedWulf) to accelerate sparse-graph problems, which are typically memory-bottlenecked on traditional x86 systems. We employ Message Passing Interface (MPI) to design an optimized scatter technique for supporting sparse-graph oriented irregular memory accesses across the cluster. We then formulate a performance model with a coefficient of determination more than 90% for understanding runtime scaling trends for our novel scatter-gather routine. We show that our ZedWulf cluster can be almost $2\times$ more energy efficient than traditional x86 based systems at processing certain kinds of sparse graph problems.

I. INTRODUCTION

In its early days, computer architects exclusively built HPC systems from specialized vector hardware; these include the Cray-I [28] and other bespoke machines such as NEC’s SX-3 and Fujitsu’s Numerical Wind Tunnel. From the early/mid 90s, x86-based systems rose in popularity due to simplicity of manufacturing and standardization of the ISA/floating-point system (Intel 8087 was an early example of IEEE-754 compliant processor hardware). Today, power density concerns and the limited impact of multi-core CPUs in HPC workloads drive the community to consider alternate configurations to squeeze out higher performance per watt [24]. Usage of multiple low powered processors, typically RISC based, in place of a single power hungry “fat” processor is surfacing as a viable alternative to curtail the energy constraint. Although this strategy gave rise to architectures such as PowerPC, ARM chips are gaining more interest in the research community [32] as in contrast to the former, ARM processors are fabricated exclusively for low powered devices.

Despite companies like Calxeda and HP promoting pure ARM-based SoCs as alternatives to x86-based server systems recently [17], these designs are mainly suited for applications constrained by communication bottlenecks rather than compute intensity. The emergence of hybrid SoCs which couple FPGA fabric with ARM processing cores, however, offer a potential entry for ARM in the HPC sector by providing low-power acceleration even for moderately compute-intensive workloads. Xilinx Zynq is one such SoC which features a dual-core ARMv7 CPU coupled with FPGA fabric. Complemented with the CPU’s ready integration of interfaces such as DRAM, Ethernet, USB, and others, thereby, facilitating the ability to host a full-fledged operating system makes it an appealing candidate for assessing the effectiveness of such a hybrid design.

![High-Level Diagram of a 32-node Zynq SoC](image)

While the present day performance gap between HPC-grade x86 processors and commercial ARM processors can be as high as an order of magnitude, large graph problems with low spatio-temporal locality can reduce the performance gap between the two architectures. Traditional CPU-based systems are bottlenecked by the inadequacy of the memory system to sustain high bandwidth during very sparse, fine-grained data accesses [23]. In this project, we study the effectiveness in using a Beowulf Cluster [38] composed of Cheap-Off-The-Self (COTS) Zynq-based ZedBoard platforms, interconnected using a Gigabit Ethernet Switch, in accelerating such sparse-graph oriented irregular computations. Figure 1 illustrates a high-level overview of the cluster setup, termed “ZedWulf” - derived from the terms Zedboard and Beowulf. Composed of 32 zedboards, this setup allows massively parallel problems to take advantage of 32 parallel memory (DRAM) lanes, and the high-bandwidth and low-latency capabilities of each of
the 32 FPGA Block-RAM (BRAM) simultaneously, thereby, showing potential for accelerating computation in tandem with Amdahl’s law.

Leveraging on this strategy and forming the basis for the recently submitted conference paper, we make the following contributions in this project:

- **Profiling the Characteristics of Zedboard** - We write and use micro-benchmarks to assess CPU, memory hierarchy and network performance.
- **Design of Optimized Global Scatter Operation** - We employ Message Passing Interface (MPI) to design an optimized scatter technique for performing an irregular scatter operation across nodes tailored for graph structures. Bearing that the zedboard saturates only 24% of the theoretical ethernet throughput for bi-directional network tests; our scatter technique manages to sustain 12% bidirectional bandwidth for irregular scatter operation between two nodes.
- **Building Performance Model** - We build a performance model for characterizing the global scatter run-time performance. The timing numbers predicted by our model correlates to that of observed performance with a coefficient of determination (R2) of 0.99.
- **Comparing performance of ZedWulf cluster against x86 Systems** - We compare the performance of ZedWulf in processing neural network simulations against performing the same on server grade x86 systems.

II. BACKGROUND

To configure and manage a variety of devices and achieve interconnection amongst them, we need to have sufficient understanding of Linux, network knowledge, good MPI proficiency and some FPGA toolset skills. This section briefly summarizes the key concepts involved.

A. Hardware

As we use the Zynq SoC on the Zedboard (Z7020) to build our experimental infrastructure, we need to understand its internal interface capabilities such as the various types of AXI interconnect buses linking the ARM Processing System (PS) block to the FPGA Programmable Logic (PL) fabric. We analyzed the performance of memory interfaces using a variety of benchmarks. Hence, we need a comprehensive understanding about the memory hierarchy structure of processors and the types of cache writing policies. Further, the former is also beneficial for calculating theoretical values of various performance figures such as MIPS and memory bandwidth for contrasting and verifying measured performance. We also hacked a Power Supply Unit (PSU) to power our cluster of 32 zedboards. Thus, it is beneficial to have experience with soldering and assembling desktop systems.

B. Linux

We set up each ZedBoard with Xillinux-1.3, a variant of Ubuntu 12.04 featuring precompiled drivers connecting the Linux host on the PS with the FPGA fabric on the PL via AXI

- Accelerator Coherency Port (ACP). Therefore, we require an intimate knowledge of the Linux OS (Xillinux) internals for understanding the interactions between PS->PL for data communication and configuration.

For multi-Zedboard setup, we configured Secure-SHell (SSH) and Network File System (NFS) services for achieving password-less shell access and file synchronization between all nodes. Thus, it is vital to understand SSH, NFS and file synchronization utility - rsync. As certain MPI operations cause the ethernet interface to fail randomly on the ZedBoard, knowledge about network debugging tools such as netstat, ethtool, iftop and ifconfig is beneficial in solving or finding a work-around to such problems.

All our experiments are scripted to be repeatable with minor modifications. Scripting was very useful to automate tedious repetitive tasks such as creating watchdog threads to monitor network status, automating installation of packages, configuring system files, and collecting benchmark results on different platforms.

Further, for achieving one of the tasks of decoding physical memory addresses from logical addresses referenced in C programs (running on PS), it is critical to understand the proc pseudo-filesystem’s map and pagemap interfaces which provide user level access to kernel pagemap related data structures.

C. Installing Packages

As not all application binary and object files are readily available for the ARM-based OS distributions, in contrast to the x86 variant, there is a basic necessity to be comfortable in resolving dependencies and building packages from source.

Further, the availability of various MPI implementations, such as OpenMPI and MPICH, and the ability to build them either for debugging or for optimal performance mandates a requisite to be comfortable with installing multiple versions of MPI outside the standard library paths and loading the required version as necessary during run-time. The same applies while linking the MPI library with MPI-dependent tools such as Valgrind Instrumentation Framework used for debugging and Intel MPI Benchmarks (IMB) suite used for profiling MPI calls, whereby we can build different versions of the latter two packages depending on the type of MPI implementation compiled with.

For building for optimal performance, MPICH is explicitly configured with –enable-fast=O3, while OpenMPI configures itself automatically for performance. For debugging, MPICH is configured with –enable-g=debug, while OpenMPI is configured with –enable-debug.

D. Network

Configuration of network plays a vital role in any cluster setup. As such, we connected each device to a common switch and configured it to have a static IP address by editing the necessary linux system file accordingly. We also updated the hostnames lookup in each device to facilitate connecting to other devices using unique hostnames. In addition to these,
the Xillinux OS assigns a constant Media Access Control (MAC) address across all the zedboards. Hence, we modified the startup script to assign a unique MAC address. Sufficient knowledge about configuration of system files is therefore necessary to achieve these tasks. Our cluster sits behind a Linksys E4200 router, running dd-wrt firmware. We set-up port-forwarding rules on the router to gain direct access to every device in the cluster by multiplexing the router’s external IP address using different ports. Thus, we require a basic knowledge about configuring routers.

E. C/C++

We used C/C++ to develop benchmarking tests and custom MPI recipes on the command-line with vim and Makefiles. Programming concepts used for writing the programs include Multi-Threading and Array Indirection. Proficiency with the C/C++ language is therefore necessary for writing efficient programs and tweaking/understanding other open-source programs. We also need to understand compiler options for purposes such as creating shared libraries and enabling/disabling certain optimizations.

Good understanding about Makefile is also advantageous not only for its simplicity in reproducing build process, but also to mitigate the dependency on additional scripts to launch MPI programs.

F. MPI

Being used as the backbone for synchronizing and message exchange between nodes running in the cluster, we need a thorough comprehension of the MPI Application Programming Interfaces (APIs) for the efficient design of graph scatter operation. These APIs include One-Sided operations such as put and get; Two-Sided Point-to-Point operations such as send and receive, and Collective Operations such as scatter and all-to-all. Knowledge about creation of User-Defined data types such as type-indexed are also crucial as they facilitate the coalescing of, otherwise, fine-grained transfers.

Familiarity with debugging tools such as Valgrind might help in decoding the ambiguous messages reported by MPI run-time in the event of errors. Further, Cachegrind utility also helps in profiling into the API calls to analyze the distribution of time spent inside any MPI function.

G. FPGA

Although this project does not intensively deal with hardware logic design, we must be coherent with FPGA design principles and methodologies, as certain micro-benchmarks require reconfiguration of the PL. These include being coherent with Xilinx ISE and Vivado Integrated Development Environments (IDE), and the ability to link custom code with Xilinx FIFO interfaces [21], both using Verilog and Vivado High Level Synthesis (HLS).

H. External Benchmarking Tools

In addition to custom written micro-benchmarks, we also use the following prewritten benchmarking suites for profiling various characteristics of the system:

- Imbench - A collection of open-source micro-benchmarks which primarily measure latency and bandwidth of memory hierarchy, and overhead of certain system calls.
- netperf & iperf - Open-source utilities to measure the network performance between two systems.
- Bonnie++ & iperf - Open-source utilities to measure the latencies and bandwidth for secondary storage devices.
- Intel MPI Benchmark (IMB) - A suit of benchmarks to analyze the performance of critical MPI communication calls and synchronization calls.

III. Literature Review

Beowulf cluster began as a low cost hobbyist alternative to state-of-art HPC systems [37]. Based on the idea of connecting relatively inexpensive COTS computers to collectively solve a particular problem, [38] was the first such cluster developed in 1994 by connecting 16 Intel DX4 processors with a 10Mb ethernet network. This eventually paved way for the creation of the first cluster based supercomputer in 1997, the ASCI Red, which employed 7,246 Intel x86 Pentium Pro processors linked using a custom-interconnect architecture [35]. Peaking the TOP500 list for nearly three years, it set out the foundation for the dominance of x86 cluster systems we continue seeing today.

The same era saw the arrival of Reduced Instruction Set Architecture (RISC) based architectures in place of Complex Instruction Set Architecture (CISC) based x86 in the form of IBM’s PowerPC processor series for usage in the embedded market [26]. Following Intel’s path of resorting to cluster design for HPC systems and exploiting the low power capabilities of the RISC architecture, IBM launched the BlueGene supercomputer series in 2004 by fusing multiple PowerPC based processors onto a single Application Specific Integrated Circuit (ASIC). Merging two PowerPC-400 (700MHz) processors onto a single ASIC and placing 16 such ASICs onto a single board to form a node, the 1st generation BlueGene/L variant [39] employed 65,536 such nodes interconnected in a 3D torus topology to dominate the TOP500 list for four years [40]. The 2nd generation BlueGene/P variant used 4 higher clocked PowerPC-450 (850MHz) cores to form the ASIC, doubled the node size to 32 ASICs and increased the network throughput by 2.5x to achieve an overall speedup of 2.8x while increasing energy usage by 60% [10]. The current 3rd generation BlueGene/Q variant employs 16 core Power A2 (1.6GHz) processor in place of the 4 core older ASIC chip, uses 5D torus interconnection topology in place of 3D and increases the throughput of each link by 2.4x [29], thereby theoretically achieving a speedup of 15x with a mere 105% increase in energy consumption [25] making it top the GREEN500 list, while excluding GPU accelerated clusters.

These prominent traits of an already existing RISC based energy efficient implementation (PowerPC) may question the importance of considering (another) RISC-based ARM architecture as a viable candidate for the HPC sector. But, ARM’s primary motive of increasing performance while improving energy efficiency in mobile devices against PowerPC’s HPC
business driven goal to increase performance per cost might potentially cause ARM architecture designs to achieve a higher performance per watt score in the future, which plays a crucial role in reaching exascale computing [24]. This statement is strengthened by the relatively much higher demand for ARM processors (in contrast to demand for HPC systems in numbers) today which helps amortize the R&D costs for the former. A recently conducted study [6] on a variety of x86 and ARM processors found the type of Instruction Set Architecture (ISA) to play a minimal role in determining the performance or energy efficiency of a particular architecture, rather, it found the performance gap between the x86 and ARM processors to merely be the result of choosing different optimization metrics while implementing the ISA. This further supports the potential effectiveness of ARM architecture in the HPC sector.

Several studies have been conducted in assessing the limitations of various COTS ARM platforms. The largest cluster setup studied [4] was the Tibidabo cluster which consisted of 192 NVIDIA Tegra-2 SoC interconnected using 1GbE network. The study concluded that the lack of high-bandwidth I/O interfaces such as 10GeE/InfiniBand and the absence of hardware support for interconnection protocols on the Tegra-2’s ARM Cortex-A9 processor are the sole limiting factors in adopting the SoC for HPC usage. On the unaltered usage of the ever-improving mobile SoCs, the study also noted that the lack of DRAM Error Correction Codes (ECC) and the coupling of unnecessary (for HPC applications) modules such as audio/video processor, display controllers and baseband processors makes out-of-box usage of such chips discouraging for HPC.

The Irdis-Pi cluster [7] is another ARM cluster composed of 64 Raspberry-Pi boards. The latter’s outdated ARMv6 architecture, further limited by the slower 100MbE network interface, caused the study to deduce this cluster to be more suitable for simple massively parallel tasks such as running prediction models based on real-time parallel sensors decoding primarily for the resulting energy savings from avoiding the usage of power hungry x86 processors for performing such trivial tasks.

A popular trend based on a similar ideology is the Microserver class of systems. Targeted for non-compute intensive applications which are bounded by I/O, such as a simple Web Server, these are stripped down versions of typical server grade systems optimized for high power efficiency at the cost of less performance [15]. Calxeda’s (defunct) EnergyCore system, featuring a quad-core Cortex-A9 processor boasting superior I/O such as SATA3 and 5x 10GbE links, was a pioneering ARM-based implementation of this concept. Other recent ARM implementations include Dell’s proof-of-concept [33], Applied Micro’s X-Gene [4] and AMD’s Opteron A1100 [30] series of microservers based on the recently introduced ARMv8 64-bit architecture (Cortex-A57). These systems can be scaled in the Beowulf fashion by simply connecting using available I/O interfaces.

While other similar studies involving clusters composed of COTS Apple-TV nodes [9] and Beaglebone-xM nodes [31] individually conclude that the systems suffer from slow compute and communication performance, the availability of SoCs which couple accelerators along with the ARM processor could likely be game changers. The study of ZCluster [20], a cluster composed of 8 ZedBoards, is one such research which considered speedup from offloading computation to accelerators. It assessed the performance of the cluster for compute intensive operations taking into account the acceleration provided by the FPGA fabric. Employing Hadoop\(^1\) software framework for distributing workloads across nodes, the study used a FIR filter based program to benchmark the cluster. The results found the cluster to provide a speedup (relative to performance on a single node) with a Strong Scaling Efficiency\(^2\) of 40% across 8 nodes. Other platforms which include accelerators include Adapteva’s Parallella boards [3], which feature a Zynq SoC along with the Epiphany accelerator [14] - 16 or 64 core RISC based processors operating at 1GHz and designed to accelerate single-precision floating point operations, and NVIDIA’s Jetson TK1 boards [2] which feature Quad ARM Cortex-A15 cores (2.3GHz) coupled with a GPU featuring 192 CUDA cores.

In contrast to the aforementioned studies which exclusively either address the limitations of the ARM processor or measure hardware accelerated speedup with reference to running benchmarks on a single node, our research work distinguishes itself by comparing the efficiency of FPGA-accelerated cluster performance against that provided by a typical server grade x86 processor, while running sparse-graph oriented neural simulations. In addition, we also provide suggestions for mitigating the limitations on the Zynq SoC taking the practical performance of other ARM SoCs into consideration.

The Message Passing Interface (MPI) is a specification providing APIs to achieve communication amongst processes running in distributed systems. Being used in almost all HPC applications [36], it has become an industry standard for programming multi-processor systems [8] since its inception in 1994 [1]. MPI usually defaults to using TCP/IP protocol on systems where higher performance protocols, such as Infiniband (IBoI), are unavailable [19]. The study performed on the Tibidabo cluster analyzed ways to improve the performance of MPI on its Tegra-2 nodes [4]. It found that using Myrinet Express message passing stack (Open-MX) in place of TCP/IP stack for generic ethernet interfaces improved the MPI bandwidth by 80% to 117MB/s, while reducing the latency by 35% to 65us. Another study explored implementing a minimalistic MPI library fully on the FPGA blocks of Xilinx Virtex2-Pro based embedded platforms as a proof-of-concept [34] demonstration. In contrast to using ethernet links however, it leveraged the MultiGigabit Transceiver (MGT) links on the FPGAs for achieving interconnection with other nodes and found the MPI latency to be 22us. Other previously

---

\(^1\)Hadoop is a framework for solving large data problems using the MapReduce paradigm in distributed computing

\(^2\)Strong Scaling Efficiency measures the ability of a program to scale as the number of participating processors are increased
discussed embedded-cluster studies [7] [9] [31] also utilize MPI for benchmarking and running applications across all the nodes. Hence, we chose to develop our graph-oriented scatter operation using MPI backend.

The unavailability of a MPI-based Scatter operation tailored for sparse-graph’s irregular scatter pattern creates a need to design a custom scatter operation using existing MPI APIs. For a similar custom need of achieving simultaneous MPI Collective Communications\(^3\) within multiple sets of processes, T. Hoefler [16] implemented the collective operations by performing multiple Point-to-Point\(^4\) communications. He also advised on the need for some scheduling strategy for conducting the point-to-point operations to avoid overloading any single node with network traffic. Given that MPI point-to-point operations can be implemented with either One-Sided or Two-Sided communication model, Jon Gibson [12] advocates on the usage of One-Sided Remote Memory Access (RMA) operations as it removes the necessity of the target process to explicitly participate in the data exchange, thereby, making the target process theoretically available for other computations and simplifying the algorithm design. However, an OpenMPI supported study [5] performed on MPI One-Sided communications revealed that the usage of RMA calls might add an implicit overhead required to reorder some operations, thereby causing additional latencies at times. A similar study supported by MPICH [18] revealed that RMA calls are often implemented on top of Two-Sided models such as Send/Recv calls, resulting in relatively higher protocol overheads and thereby losing out to pure Two-Sided communication calls. This study, in tandem with another study based on the new MPI-3 standard [11], also noted that for interconnection architectures supporting Remote Direct Memory Access (RDMA) such as InfiniBand, performance of RMA can potentially surpass that of Two-Sided communication by up to 70%.

**IV. IDEA**

Many applications in Physics and Biological domain, such as optimization and simulation problems, often inherently use Graph algorithms [27]. A Graph structure represents a collection of graph nodes and the interconnecting edges. Each node performs some computation on data received from incoming edges and passes the output of computation to its outgoing edges, as shown in Figure 2. Typically, each node is connected to at least one edge.

Graph processing can be viewed as a repeated two-stage process, viz. Compute & Update Edges, each of which can be parallelized as illustrated in Figure 3. Irregular memory access pattern while updating the edges often bottleneck traditional x86-CPU based implementations of large graph processing [23]. We circumvent this problem by leveraging on the low-latency capabilities of FPGA BRAMs on the Zynq SoC. However, the inability to fit all the graph nodes on the limited BRAM resources poses to be a hurdle. We attempt to use multiple Zynq SoCs as individual Processing Elements (PE) to try and accommodate the graph nodes in multiple PEs, thereby effectively multiplying the available number BRAM memory lanes (Figure 4). This requires partitioning the graph structures. Irrespective of the efficiency in partitioning, it almost always brings about edges which connect to nodes present in other PEs (henceforth called Remote edges). Figure 5 shows an illustration of this behavior.

Unlike Local edges, which connect nodes present within the same PE, updating remote edges is typically an order of magnitude slower as the data needs to be transferred from the origin PE to the target PE using the ARM CPUs to handle network packet transfers. Thus, in contrast to Figure 4, we are unlikely to see an \(\alpha\) times speedup from using \(\alpha\) PEs.

The process of performing edge updates is essentially a Global Scatter operation, whereby each PE scatters the re-
quired node values amongst all PEs (including itself for local edges). To maximize the performance gain while using such ARM-FPGA hybrid distributed systems, there is an inherent need to reduce the time spent in fulfilling the global scatter operation. We propose an optimized design of the global scatter technique using the Message Passing Interface (MPI) library.

A. Representation of Graph Structures

The scatter operation is a simultaneous operation of exchanging values between nodes as dictated by the edges. To facilitate an efficient implementation, it is necessary to store the required graph information in a specific manner. Listing 1 shows the psuedocode of the scatter operation in the simplest case.

This method of having the origins and targets of each edge point to a common memory location (\texttt{node[]} in Listing 1) has the following issues:

1. For nodes with at least a pair of incoming and outgoing edges, an incoming message might overwrite the node buffer before the latter is transferred to other dependent nodes (outgoing edges).
2. For nodes with multiple incoming edges, the messages from multiple edges might overwrite the node buffer.

To avoid such data inconsistencies, it is necessary to have the origin and target of each edge point to different sets of memory locations.

Nodes

Bearing that in mind, for a graph system consisting of \( V \) nodes and \( E \) edges, we represent the nodes using two arrays:

\begin{itemize}
  \item \texttt{node\_memory} - Stores the value held by each node. This value is typically the result of some computation performed on incoming edges. Each element of the array pertains to exactly one unique node.
  \begin{verbatim}
  node\_memory[i] = <node\_i>
  len( node\_memory[] ) = \( |V| \)
  \end{verbatim}
  \item \texttt{message\_memory} - Acts as placeholder for incoming edges. Unlike the \texttt{node\_memory}, each node does not necessarily own equal number of elements in the array. Thus, we use another array (say \texttt{blklen}) of length equal to the total number of nodes to specify the number of memory locations attributed to each node.
  \begin{verbatim}
  message\_memory[i] = <NULL by default>
  len( message\_memory [] ) = \( |E| \)
  blklen[i] = <node\_i.num\_incoming\_edges>
  len( blklen[] ) = \( |V| \)
  \end{verbatim}
\end{itemize}

Fig. 5: Locality of Edges after Partitioning

Fig. 6: Representation of Nodes using Arrays

Figure 6 illustrates the data structures for representing the nodes using nodes with 3, 1 and 2 incoming edges as examples.

Edges

Edges represent a connection pattern which exists between the origin (\texttt{node\_memory [i]}) and the target (\texttt{message\_memory [i]}) arrays. We represent the edges in two formats depending on the locality (local vs remote edges) -

1. Compressed Sparse Row (CSR) format for local edges, and
2. a format similar to storing the origin and target displacements of each edge in two arrays (henceforth called Edges Stored as Displacements – ESD format) for remote edges. Figure 7 compares both the representation formats taking the graph structure in Figure 5 as an example. We chose this strategy for the following reasons:

\begin{itemize}
  \item \textit{Remote Edges} - Despite the higher storage efficiency of CSR format, we adopted ESD here as the MPI APIs require a format specifying displacements for source and receive buffers (section C). The ESD uses two arrays, viz. \texttt{origin} and \texttt{target}, to store the origin and target addresses of each edge sequentially.
  \begin{verbatim}
  target[i] = <edge\_i.target>
  \end{verbatim}
\end{itemize}
ii **Local Edges** - Using CSR format has better Space and Memory Read complexity than ESD for all cases when |E| > |V|. As the number of edges is usually more than the number of nodes for a typical graph structure, CSR is an optimal fit for representing local edges. The CSR representation describes edges using two arrays, viz. origin_i and target. The target array stores the target of each edge, while the origin_i (origin index) is used to attribute elements of the target array to each origin node. In other words, for every index (say i), i acts as address to node_memory, while the elements from target[ origin_i[i-1] ] up till target[ origin_i[i] ] represent the target addresses.

The addresses stored in origin and target arrays are essentially displacements pointing to the node_memory and message_memory arrays.

---

**B. High-Level Global Scatter Design**

---

![Fig. 8: Overview of Global Scatter](image)
MPI is a standardized message passing system which defines a set of APIs for achieving communication and synchronization amongst processes running on distributed systems. Since the MPI specification does not provide a direct function to perform a sparse-graph-oriented scatter, we use existing functionality provided by MPI to achieve our cause. With reference to Figure 8, the tasks required for implementing the Global Scatter can be grouped into two phases, viz. Initialization and the actual Scatter phases. The Initialization phase is a one-time preparatory phase which comprises of the following stages:

i **Generate Graph** - Creates a sample graph structure randomly or from an mtx data file.

ii **Partition and Generate ESD Pairs** - For a graph partitioned into n PEs, create ESD for all possible pair-wise combinations of PEs (self-included) in-order for every PE to achieve scatter operation with every other PE.

iii **Distribute ESD Pairs** - Distribute the generated ESD arrays to the respective PEs.

iv **Build Optimized Data Types** - Create optimized data types for achieving the scatter operation from the received ESD arrays.

A single process (typically the Master process) executes the first three stages, while all other processes (Slaves) execute the last two stages. Following the initialization phase is the Scatter phase, where all scatter operations take place. It consists of two possibly overlapping operations:

i **Local Scatter** - Scatter operation which concerns only edges local to each PE, i.e. the origin and target addresses fully point to memory locations within the PE.

ii **Remote Scatter** - Scatter operation which concerns all remote edges, i.e. the origin addresses are local to each PE, while the target addresses reference memory locations inside other PEs.

C. Initialization

We generate a random graph structure of required number of nodes and edges using legacy code. Partitioning the graph implies dividing the contiguous node_memory and message_memory arrays into smaller chunks. Therefore, the edges’ origin and target arrays, which are pointing to the un-partitioned address spaces, need to be updated to reference to the partitioned address spaces. To identify the target address space, we add a target_pe array to store the target PE of each edge. Figure 9 demonstrates the sequence of operations performed taking the graph structure in Figure 7 as an example.

![Fig. 9: Operations Peformed to Partition a Graph Structure](image)

Although we have retained the graph structure after partitioning using an additional target_pe array, this representation does not yield an efficient scatter operation. Consider the global scatter pseudocode shown in Listing 2.

There is an if condition for every processing of an edge to decide between whether to perform local or remote operation. Further, for every remote edge, we have to call a function to do remote transfer (send_to_pe) separately. This adds network overhead; rather being limited by network bandwidth, this approach would limit us by network latency. To rectify the flaw, we make use of multiple ESD structures, used for storing the origin and target displacements for each target PE separately, to perform coalesced scatter operations such as the psuedocode shown in Listing 3. Here, instead of calling send_to_pe function separately for each edge, we call a modified function send_multiple_to_pe for every remote PE.

Listing 2: Global Scatter Psuedocode using target_pe

```
for every origin_dis, target_dis, PE in (origin, target, target_pe):
  if PE is myRank:
    # do local scatter
    message_memory[target_dis] = node_memory[origin_dis]
  else:
    # do remote scatter
    # (to which PE, what to send, (displacement at target PE))
    send_to_pe (PE, node_memory[origin_dis], target_dis)
```

Listing 3: C Code Listing for Distributing node_memory and message_memory Lengths across all MPI Processes
The latter function uses arrays of source and target displacements to perform a coalesced transfer to a particular target PE. We can seamlessly implement such a function leveraging on existing MPI APIs. We generate ESDs for all pairs of PEs by utilizing the `target_pe` array, as illustrated in Figure 10. Each ESD consists of two arrays:

- **origin[n]** - Array containing displacements of values to be sent to PE n.
- **target[n]** - Array containing displacements of values to be received by PE n.

![Fig. 10: Operations Performed to Partition a Graph Structure (Continuation from Figure 9)](image)

For every PE in a system of n PEs, we generate n ESD pairs for describing scatter operation originating from each PE to n other (including itself) PEs separately. Thus, this gives rise to $n^2$ ESDs in total.

The Master process performs the sequence of operations (discussed so far) from generating the graph until creating the ESDs between every PE-PE combination. Since the PEs are mapped to unique Slave processes, we need to distribute all data structures to the respective MPI processes. Listing 3 shows the first stage of this process, where we perform MPI Scatter operations (Figure 11) to distribute the `node_memory` and `message_memory` arrays across all processes. The distribution of ESD structures, however, depends on the type of MPI Communication paradigm chosen. MPI offers two types of communication paradigms, viz.:

i **Two-Sided Communication** - For a point-to-point transfer, both the sender and receiver processes synchronize and hand-shake for the communication to occur. The sender calls `MPI_send()`, while the receiver calls `MPI_recv()`.

```c
MPI_Send(
  void *src_buf,
  int count,
  MPI_Datatype datatype,
  int targetRank,
  int tag,
  MPI_Comm comm)
```

```c
MPI_Recv(
  void *dst_buf,
  int count,
  MPI_Datatype datatype,
  int originRank,
  int tag,
  MPI_Comm comm)
```

ii **One-Sided Communication** - For a point-to-point transfer, only a single process initiates the communication - the sender (in the case of `MPI_put`) or the receiver (in the case of `MPI_get`). There is no synchronization in this approach as the other process does not explicitly participate in the communication.

```c
MPI_Put(
  void *origin_buf,
  int origin_count,
  MPI_Datatype origin_datatype,
  int targetRank,
  int target_disp,
  int target_count,
  MPI_Datatype target_datatype,
  MPI_Win win)
```

For the case of one-sided communication, it is necessary for the origin process to specify both the origin and target displacements (in the form of `MPI_Datatype` structures). Contrastingly, for two-sided communication, the origin process specifies the origin displacements, while the target process specifies the target displacements. Thus, we can conceive that the context of `target[]` displacements arrays differ between both the communication styles; Table I lists the differences.

Assuming we store the ESDs (previously shown in Figure 10) in the following format, Listing 4 shows the psuedocode for distributing the `origin` and `target` arrays across all MPI processes for both One-Sided and Two-Sided based

```c
MPI_Send(
  void *src_buf,
  int count,
  MPI_Datatype datatype,
  int targetRank,
  int tag,
  MPI_Comm comm)
```
TABLE I: Semantics of target[] Arrays for One-Sided &
Two-Sided Communication

<table>
<thead>
<tr>
<th>One-Sided</th>
<th>Two-Sided</th>
</tr>
</thead>
<tbody>
<tr>
<td>origin[i]</td>
<td>Contains displacements for specifying what to send to target process i</td>
</tr>
<tr>
<td>target[i]</td>
<td>Contains displacements for specifying how to handle sent over data stream at the target process i</td>
</tr>
</tbody>
</table>

Communication paradigms (the actual C code Listing omitted to hide increased complexity):

- origin[i][j] - Contains displacements for building the data stream at process i, intended to be sent to j.
- target[i][j] - Contains displacements for processing data stream at process j, received from process i.

After distributing the origin and target arrays accordingly, each node converts the received ESD structures into MPI-friendly representation using the MPI_Type_indexed API. This API accepts as input a list of displacements and block lengths of each displacement as input. We provide a series of 1s as block lengths for the displacements origin[] and target[]. The new structure then gets finalized using the MPI_Type_commit API. Figure 12 shows an illustration of converting a target array into a MPI data type. With the exception of ESD for local scatter, we convert every other origin and target arrays into MPI data types (origin_datatype and target_datatype structures) in a similar manner. For local scatter, we convert the corresponding origin and target arrays into CSR format (shown in Figure 7).

This concludes the initialization phase, facilitating the commencement of Scatter phase.

D. Local Scatter

The local scatter operation can be performed using array indirect assignments using the origin and target displacements specified in the CSR format. Listing 5 shows the corresponding pseudocode.

Listing 5: Local Scatter (CSR format) Pseudocode

Optimization

Although CSR is efficient, it is only efficient when number of edges is more than number of nodes. For a system containing many MPI processes, the amount of local edges in each process will likely be minimal in comparison to the total number of nodes in each process. In such cases, it would be inefficient to iterate across all |V| elements in origin_i (Listing 5 - lines 4, 5) when we know that very few nodes will actually contain outgoing local edges. Thus, we compress the origin_i array, by adding another array - origin. Figure 13 illustrates this optimization, while Listing 6 shows the updated pseudocode for local scatter.

Listing 6: Local Scatter (Optimized CSR format) Pseudocode
E. Remote Scatter

Given that MPI allows two types of communication paradigms, we implement one-sided and two-sided based versions to compare the performance.

One-Sided Remote Scatter

One-Sided Communication is synonymous with Remote Memory Access (RMA) operations. Given that there are two types of RMA operations - put and get, we chose a put-based implementation as it has lesser servicing overheads than an equivalent get operation. Each process first exposes its node_memory array for remote accesses by creating a MPI Window using MPI_Win_create API; it also creates a leeching window for every other host window created in other MPI processes as shown in Listing 7. Thus, every process has 1 window permitting remote access from other nodes and n-1 windows (for a system of n processes) for allowing remote accesses to other processes. Creation of MPI windows is part of the initialization phase as it is a one-time operation.

Listing 7: Pseudocode to Create MPI Windows

We then use Passive Target Communication strategy for gaining remote access. This approach uses MPI_Win_lock and MPI_Win_unlock functions, for opening and closing RMA sessions (also known as RMA access epoch) with other processes. In a naive implementation, each process will start an RMA access epoch with win lock, call a put operation, and close the epoch with win unlock sequentially for every process, including itself. Scrutinizing the code and the run-time performance, we uncovered opportunities for improving performance. We overlap communication in the system by (1) having simultaneous epoch sessions in progress and (2) ensuring load-balanced distribution of message transfers by scheduling data transfers in a cyclic fashion. In this optimized approach, a global scatter operation consists of each process starting simultaneous epoch sessions with all other processes first, followed by calling coalesced put operations on every process and finally closing the epochs in a cyclic fashion. Here, cyclic fashion implies that each process closes the epoch in an orderly manner starting with the process next to it. In contrast, having all processes close the epoch sessions in an identical fashion will result in the MPI protocols rushing to complete network transfers originating from all processes towards a particular process, before moving on to repeat the same on the next process; this will yield low scatter throughput due to overloading of network packets on each process. We start each epoch using MPI_LOCK_SHARED flag to permit multiple processes in locking the same memory region for allowing parallel put operations on the same window. Listing 8 shows the pseudocode for one-sided remote scatter.

Listing 8: Pseudocode for One-Sided Scatter

When using MPI_Put to allow RMA-based distribution of data, since all arguments specifying origin and target buffer is provided by the origin process, there’s an inherent need for the MPI protocols to encode the target data type as a metadata. Thus, reducing the size of metadata is critical to maximize performance. With reference to Figure 12, as the target data type is a fusion of both target and blklen arrays, we optimize the origin and target arrays in such a manner that reduces length of the blklen array. We apply this optimization on all ESD pairs at every process before building the MPI data type. Figure 14 illustrates this optimization. This strategy potentially reduces the size of MPI metadata transferred to the remote process.

Fig. 14: Optimizing Target Array for Reduced Length

Nevertheless, as only the process initiating the put calls the MPI API, there’s an inherent need for the MPI protocols to encode the receive data type as a metadata each time resulting in unnecessary network communication overheads.

Two-Sided Remote Scatter

By resorting to MPI two-sided communication using MPI_SendRecv API, we rectify the drawback in One-Sided implementation by having both the origin and target process specify the send and receive buffers in every matching MPI call. MPI_SendRecv is a bidirectional point-to-point operation which facilitates the calling process to send data to another process while simultaneously receiving data from
another process. Figure 15 shows the analogy between one-sided put and two-sided sendrecv functions.

![Diagram of one-sided put and two-sided sendrecv functions](image)

Fig. 15: Put vs Sendrecv for a 1->2->3->1 Transfer Pattern

Using sendrecv allows us to mitigate the need for MPI to send the target data type as a metadata, as we can directly supply the target data type to the matching recv call in the receiving process itself. Unlike RMA based put, two-sided communication does not require starting epoch sessions. But, in contrast to put, where the order of put operation is irrelevant for the completion of the scatter operation, the risk of deadlocks is eminent for two-sided approach due to the blocking nature of the sendrecv call. Therefore, we schedule the send-receive operations in a manner to eliminate deadlocks fully, while also scheduling data transfers in a manner to avoid network contention across nodes. The corresponding pseudocode for remote scatter is shown in Listing 9.

![Listing 9: Pseudocode for One-Sided Scatter](image)

V. Experiments

In this section, we report and analyze the performance characteristics of the ZedWulf cluster by running a set of custom and open-source benchmarking utilities. We assess the performance of the global scatter operation across varying number of MPI nodes and model the scatter run-time performance. We finally perform neural simulations powered by our global scatter routine on the ZedWulf and contrast measured performance against several x86 systems.

A. Hardware Setup

ZedWulf is a cluster composed of 32 Zedboards, interconnected using a Netgear GS748T 48-port Gigabit Smart Switch. With a rated switching capacity of 96Gb/s, the switch can sustain 2Gb/s duplex bandwidth per 1GbE ethernet link connecting each zedboard. We powered the system using a Seasonic Platinum 1000W PSU, by leaching power from a single PCIe EPS12 power rail with appropriate fuse protection. We stacked the zedboards on top each other and provided cooling from 2 fans placed on either sides of the stack as shown in Figure 16.

![Fig. 16: ZedWulf Setup](image)

B. Software Stack

We configured each Zedboard to run Xillinux-1.3 [22], an Ubuntu-12.04 based Linux distribution which comes coupled with drivers to communicate with the PL using AXI-ACP. We built software packages such as MPI libraries and benchmarking kernels with gcc-4.6.3 along with appropriate optimization flags enabled. For the purpose of syncing files across all 32 nodes, we used Network File System (NFS) to mount a common storage medium across all zedboards. We hosted the NFS server on the Master node - a zedboard. Figure 17 displays the software stack of each node.

![Fig. 17: Software Stack](image)

C. Microbenchmarking the ZedWulf

We ran a variety of micro-benchmark kernels to identify the performance limits of the memory hierarchy, CPU-FPGA link, secondary storage mediums and network.

Memory System

Figures 18 and 19 show the latency and bandwidth numbers of the memory system, which we generated using mem_lat_rd and bw_mem utilities from lmbench-3.0 benchmarking suite.
As expected, we observe L1 cache access to have the lowest read latency of 6ns, followed by the L2 cache which is 6.5x slower and the DRAM which is 11x slower, each relative to L1 cache. The sequential read bandwidth also follows the previous trend making the L1 cache the fastest at 7.4TB/s, followed by the 5x slower L2 cache and the 11x slower DRAM. We, thereby, make the following observations:

i. L1 and L2 writes are 11% faster than their read counterparts.
ii. L1-Read, L1-Write, L2-Read, L2-Write and DRAM-Read bandwidths double upon running 2-core benchmark, relative to 1-core results. This shows that each core is not bandwidth limited for (1) read and write operations in the caches and (2) read operations in the DRAM.
iii. DRAM-Write net bandwidth barely increases by 4% while both the cores are writing, relative to when only either core is writing. This suggests that for application kernels with very heavy memory write operations such as while populating a large sequence of numbers, a multi-threaded version would perform only as good as a single-threaded (sequential) variant.

![Fig. 18: Summary of Memory System Latencies](image)

Fig. 18: Summary of Memory System Latencies

While the previous two plots showed the best case (bandwidth) and worst case (latency) numbers, we compare the real world performance between performing sequential reads and performing random reads for varying read data sizes in Figure 20. For both the experiments, we used array indirection to supply the read addresses; in the case of sequential benchmark, the addresses are sequentially incremented while in the case of random benchmark, the addresses are randomly generated to point to some location in a 64MB array stored in the DRAM.

We observe that sequential access bandwidth saturates at 620MB/s in accordance with the result reported by lmbench (Figure 19) while, the random access bandwidth constantly averages at around 36MB/s corresponding to a latency of around 22ns/byte. This number is 3x lesser than the DRAM latency of 66ns reported by lmbench as we fetch 4 consecutive bytes for every address given in our benchmark. We chose to fetch 4 bytes per given address as programs typically work with int/float datatypes which are of length 4 bytes. Fetching 4 bytes/address is, therefore, observed to be 3x faster than fetching 1 byte/address, as the last 3 bytes are already fetched into the cache in the former’s case. Thus, we deduce that DRAM random accesses are around 17x slower than performing sequential accesses.

**AXI-ACP Interface**

The Xillinux drivers leverage the AXI-ACP interface to communicate data between the CPU and the FPGA. We perform a loop-back test on the AXI-Accelerator Coherency Port to measure the latency and bandwidth of the link. We conduct the test in three variations:
i Sequential - Perform a write, then a read operation on a single thread.

ii Multi-Threaded (Single Core) - Issue a write on one thread and a read on another thread, while pinning both the threads to a single core.

iii Multi-Threaded (Multi-Core) - Similar to (ii), but each thread is pinned to a unique core.

From the results plotted in Figure 21, we observe that multi-threaded implementations are consistently faster than sequential implementations. Time shared multi-threading (pinned to single core), has 16% lower latency than truly parallel multi-threading (pinned to unique core) for sizes less than 42KB. We attribute this behavior to the overheads in servicing L1 cache misses, which is more apparent for small data sizes. As the data size increases, truly parallel variant begins to be consistently faster by 8% than the time shared variant. Thus, appropriate implementation style should be used depending on the PS-PL communication density demanded by the application.

In summary, considering the best results from the three implementation variants, ACP has a latency of 55ns while sustaining an average bandwidth of 380MB/s for sequential transfers ranging beyond 64KB in size.

Secondary Storage

With the Zedboard having a SD card and USB interface, we run tests to assess the efficiency of different I/O devices. We run bonnie++ and ioping benchmarks on the following storage mediums:

i SD Card - A SanDisk Ultra Class 10 SD card connected directly to the onboard SD slot, on which the OS is hosted upon.

ii SSD - A Samsung 840 Pro Solid State Drive (SSD) connected via the USB2 interface using a high performance SATA to USB3 adapter.

iii HDD - A Western Digital Black Hard Disk Drive (HDD) connected in a similar fashion as (ii).

From the analysis of latencies and bandwidths of various storage solutions (summarized in Figure 22), we make the following observations:

i The SSD, connected using a USB2 interface, has 28% lower read latency and 20x lower write latency than the native SD card.

ii USB2 specifications limit the bandwidth performance of the SSD. Assuming a theoretical peak bandwidth of 280Mbps for the USB2 interface, zedboard only manages to achieve 75% efficiency with the current setup.

iii HDD, despite having higher latencies than both SD and SSD, manages to sustain higher read and write bandwidths than the native SD card.

We hence note that I/O intensive applications will benefit from USB-attached SSD solution in comparison to the native SD card medium.

Network

We leverage the netperf and iperf network utilities to find the characteristics of the zedboard’s network link by measuring performance when the target is another zedboard and comparing it with when the target is an x86 machine.

Results from Table II show that all network transfers (excluding ping) are faster between Zedboard and x86 PC, rather than between two Zedboards. We draw the following observations from the results:

i The current setup of Zedboard achieves 54% theoretical throughput with unidirectional TCP and 59% theoretical throughput with unidirectional UDP, when the target is
**TABLE II: Network Performance Comparing Zed - Zed and Zed - x86**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Zedboard - Zedboard</th>
<th>Zedboard - x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ping</td>
<td>0.166 ms</td>
<td>0.230 ms</td>
</tr>
<tr>
<td>Unidir - TCP</td>
<td>481 Mbps</td>
<td>544 Mbps (1.13x)</td>
</tr>
<tr>
<td>Unidir - UDP</td>
<td>375 Mbps</td>
<td>588 Mbps (1.56x)</td>
</tr>
<tr>
<td>Bidir - TCP</td>
<td>350 Mbps</td>
<td>450 Mbps (1.28x)</td>
</tr>
</tbody>
</table>

a faster machine (x86). Nevertheless, TCP throughput reduces to 23% (450/2000) when in the case of bidirectional transfers.

ii Between a pair of zedboards, the achieved unidirectional network throughput decreases to 48% for TCP and 38% for UDP. The efficiency for bidirectional TCP throughput also decreases to 17.5% (350/2000).

This tells us that MPI network operations cannot possibly exceed 60MB/s (48% efficiency) for unidirectional point-to-point operations between zedboards and that for bidirectional operations such as sendrecv(), the performance might be further dampened to 43MB/s (18% efficiency).

**D. Profiling MPI**

OpenMPI and MPICH are the two popular implementations of the MPI library which are available for the ARM architecture. We benchmark both the implementations to choose an optimal version for running our global scatter operation. We compiled the results in this section with custom MPI profiling kernels and with the use of Intel MPI Benchmark (IMB) suite. We first profile, using 2 MPI processes each on a unique zedboard, the latencies of commonly used one-sided, two-sided and synchronization APIs as listed below:

i **Put (Bidir)** - Process A executes put on process B, while process B executes a similar put on process A simultaneously.

ii **Put (Unidir)** - Process A executes put on process B.

iii **Sendrecv (Bidir)** - Process A and B call sendrecv on each other

iv **PingPong (Unidir)** - Process A executes send to process B, while the latter calls recv.

v **Barrier** - A blocking synchronization operation which will forbid progress until both process A and B has reached that portion of code executing the barrier operation.

From the results of the latency measurements in Figure 23, we can observe that MPICH has lesser overhead for all operations than OpenMPI. In a similar manner, we proceed to compare the bandwidth rates for one-sided and two-sided bidirectional operations in Figure 24. We note the following observations from plots in figures 23 and 24:
i Bidirectional operations, in general, have higher latencies than their unidirectional counterparts (Put-Bidir vs Put-Unidir & Sendrecv vs PingPong).

ii Two-Sided operations, in general, have lower latencies than One-Sided operations (Put vs Sendrecv/PingPong).

iii MPICH not only has lower latencies for all operations, but also sustains higher bandwidth than OpenMPI for all bidirectional operations most of the times (Figure 24).

iv Although MPICH has lower latencies for Two-Sided operations than One-Sided (ii), for larger data sizes, sendrecv and put roughly perform the same (Figure 24).

v Despite netperf reporting only 43MB/s for bidirectional bandwidth, MPICH implementation manages to hit 60MB/s for both large sized bi and uni directional transfers (Figure 24).

As a final test to assess how CPU resource savvy MPICH and OpenMPI are, we measure the performance drop in performing a two-sided communication operation when one of the two available cores is unavailable (busy performing some computation). Figure 25 shows the difference between idle and 50% loaded system states.

![System Idle vs System 50% Loaded](image)

Fig. 25: Idle vs Loaded System States

We calculate the resulting speed drop using the following equation and plot the results in Figure 26:

\[
1 - \frac{\text{Bandwidth}_{\text{Under Load}}}{\text{Bandwidth}_{\text{Idle}}} \quad (1)
\]

The lower the speed drop, the less resource savvy a MPI implementation is deemed to be. From Figure 26, we observe MPICH to perform better in this category too. Thus, we chose MPICH as the MPI implementation of choice to run our scatter operation with.

Besides profiling One-Sided and Two-Sided operations, it is also important to analyze the time taken for a global MPI_Barrier operation for varying number of MPI processes. Figure 27 plots the range of time taken for every barrier operation as a function of number of participating processes. We varied the number of processes between 1 and 26 (not 32 due to lack of boards at the time of benchmarking).

The result shows that the average time taken for the barrier operation increases as the number of processes increases. Further, it also hints that the worst case performance of the operation worsens as the number of processes increases; we attribute this behavior to the increased probability of jitter in network as the number of participating processes increases.

E. Summary of Characteristics

Figure 28 summarizes the latencies of the key components of the system. The memory system has exponentially lower latencies in the nanoseconds region, followed by the ACP having a 71us latency and tailed by network-based MPI functions in the 110us region. Compared to DRAM, network access overheads seem to be 104x slower; however, this is not always the case as shown in Figure 29. Once MPI_Put (purple) gets to transfer sufficient amount of data (100KB), it seems to overtake the DRAM’s random access bandwidth (green). This is an interesting result which tells that at times, fetching something from remote memory can be faster than fetching from local memory!
F. Global MPI-Based Scatter

The Global Scatter, discussed in Chapter IV, is a combination of local and remote scatter operations tailored towards achieving edge updates for a graph structure distributed across MPI processes. We first run the scatter benchmark using a modified graph generator to vary the average number of remote edges per PE. Figure 30 shows a contour plot where the data size (number of edges) and percentage of remote edges are kept variable, while the time taken for a 32-node (MPI process) scatter operation is plotted. We deduce that as the percentage of remote edges increases, the total time taken for the scatter operation to complete also increases. We attribute this to the much slower network access, in comparison to local memory accesses.

Next, we evaluate the performance of various optimizations performed in our One-Sided MPI-based scatter design by running different variants of our scatter implementation using an actual graph structure (att-twotone.mtx). Figure 31 plots the performance across varying number of MPI Processes.

The base version (seq_no_sched) uses a sequential approach for performing put without any scheduling to avoid network contention. This means that only one epoch session per process will be open at any given time, and all processes will be sending data to a common node, before moving on to the next common node. The 1st optimization (parallel_no_sched) improves on the former by having multiple epoch sessions per process in progress at any given time. Nevertheless all epoch sessions are closed in an identical manner as no scheduling exists. This implementation improves on the former by around 5% to 23%. The 2nd optimization corrects the previous flaw by including a scheduling strategy to mitigate network contention. This improves performance from 2% for 2 MPI processes to 52% for 32 MPI processes. This is so, as the lack of scheduling has a linearly increasing impact as the number of participating processes increases. All optimizations so far
are using MPI APIs to perform local scatter. This potentially amounts to unnecessary overheads. The final optimization (parallel_with_sched_local_scatter) rectifies this flaw by using the CSR format to perform direct array indirect assignment as shown in Listing 5. This boosts performance by 2% for 32 MPI processes up to 50% for 2 MPI processes. We see a decreasing effect of performance gains as an increasing number of processes (PEs) corresponds to lesser % local edges; hence, the optimization done for local scatter fades away relatively.

We compare the optimized one-sided put based implementation with two-sided sendrecv based implementation in Figure 32. The sendrecv implementation also features all applicable optimizations previously mentioned. Results, while comparing the performance for 16 MPI processes, show that the sendrecv implementation improves performance achieved from put-based implementation by more than 2x on an average. This huge gain is possible as (1) our setup lacks support for Remote Direct Memory Access (RDMA) operations and (2) the sendrecv abandons the need to encode target data type as metadata. Nevertheless, we expect the put based implementation to perform equally well (if not faster) for architectures supporting RDMA, such as the Infiniband-QDR.

**G. Modelling Global Scatter**

The time taken for a global scatter operation is essentially the sum of individual time taken for local and remote scatter operations, separately.

**Remote Scatter**

Through repeated experiments, we found that remote scatter operations can be modelled as a function of transfer size (W) and number of participating MPI processes (P) using the following equation:

\[ T_{MPI}(W, P) = L_{MPI}(P) + \frac{W}{BW_{MPI}(P)} \]  

Here, \( L_{MPI}(P) \) represents the latency of a remote scatter operation, while \( BW_{MPI}(P) \) denotes the average scatter bandwidth between processes associated with a system of \( P \) processes. The values \( L_{MPI}(P) \) and \( BW_{MPI}(P) \) are calculated on sampled observations using appropriate data fitting tools. Figure 34 plots \( L \) and \( BW \) for all \( P \) ranging from 2 to 32, calculated for One-Sided Global Scatter.

**Local Scatter**

With reference to the psuedocode for Optimized CSR-based local scatter (Listing 6), based on the iteration structure of the loop, we can deduce that the target, origin and origin_i arrays are all accessed sequentially. This corresponds to a sequential read operation of total length equal to thrice the number of local edges \( (3M_L) \), at the worst case. Fetching of those three arrays from DRAM, over time, will give rise to cache misses. If the DRAM has a latency
of $L_{MEM}$ and a sustained bandwidth of $BW_{MEM}$, we can model the time spent in reading the three arrays using the following equation:

$$3M_L \times \alpha_0 L_{Mem} + \frac{1 - \alpha_0}{BW_{MPI}}$$  \hspace{1cm} (3)$$

The last array, node memory, is accessed indirectly using the displacements provided by the former three arrays. The irregular access pattern gives rise to a potentially higher percentage of cache misses ($\alpha_f\%$), relative to $\alpha_0\%$. In the worst case scenario, $M_L\%$ elements will be read; we model the time spent in reading this array, in a manner similar to (3), as follows:

$$M_L \times \alpha_1 L_{Mem} + \frac{1 - \alpha_1}{BW_{MPI}}$$ \hspace{1cm} (4)$$

The final equation for characterizing local scatter is (3) + (4):

$$T_{Mem}(M_L) = \left[3M_L \times \alpha_0 L_{Mem} + \frac{1 - \alpha_0}{BW_{MPI}}\right] + \left[M_L \times \alpha_1 L_{Mem} + \frac{1 - \alpha_1}{BW_{MPI}}\right]$$

**Combined Global Scatter**

The total time taken for the global scatter operation is given by the sum of $T_{Mem}$ and $T_{MPI}$. Percentage of remote edges ($R$) defines what portion of edges belongs to remote scatter, with the rest belonging to local scatter. For a system of $P$ PEs, each composed of $M$ edges/PE, along with $R\%$ average remote edges per PE, we compute the total time using the following equation:

$$T_{Scatter}(M, R, P) = T_{MPI}(M \times R, P) + T_{Mem}(M \times (1 - R))$$

The challenge lies in predicting the right value for percentage of cache misses ($\alpha_{textsubscript0} \& \alpha_{textsubscript1}$) and the percentage of remote edges ($R$). $R$ can generally be computed prior to or during run-time. Figure 35 shows the time predicted by this model (with heuristically guessed $\alpha_{textsubscript0}, \alpha_{textsubscript1}$ & $R$) against observed results for one-sided global scatter operation. We measured this model to have a $R^2$ coefficient of determination of 0.99, proving high coherence to real-world results.

**H. Performance of ZedWulf vs x86 Systems**

In this section, we show the performance of our ZedWulf\textsuperscript{5} cluster relative to several server grade x86 systems. We employ Neural Simulations as a sample sparse graph problem to perform Bulk-Synchronous Parallel (BSP) style computations based on our Global Scatter routine. We compare our performance against numerous x86 systems running fully optimized neural simulations directly across multiple (4-6) OpenMP threads. We use the MTEPS (Millions of Traversed Edges Per Second) metric to quantify the processing performance of the systems. We also measure the total power of each system.

\textsuperscript{5}Although we termed ZedWulf as a cluster of ZedBoards, micro-ZedBoards were used in place of the former due to lower energy consumption. Micro-ZedBoards have identical architecture as the ZedBoards with the difference being the absence of peripherals found on the ZedBoards. This change was incorporated only during the final stages of this project upon the arrival of the micro-ZedBoards arrived.
(including the Network switch for the ZedWulf cluster) while running the benchmark. Although we termed ZedWulf as

![Fig. 36: Performance-Power tradeoffs for x86 Systems against ZedWulf](image)

From the Power vs MTEPS plot as shown in Figure 36, we can see that ZedWulf has indeed surpassed the performance of all the x86 systems for graphs with 32M nodes and 32M edges. ZedWulf performs 16% faster than the 6-core E5-1650 and 36% faster than the 4-core i7-4770 while consuming lesser power than either of the x86 systems. This observation is further bolstered by figure 37 which compares the energy efficiency of all the platforms. Here, ZedWulf exceeds the energy efficiency of E5-1650 and i7-4770 by 46% and 38% respectively.

![Fig. 37: Energy Efficiency comparisons between x86 Systems and ZedWulf](image)

VI. CONCLUSION

In this project, we have successfully configured and profiled various characteristics of the 32-node Zedboard based cluster. From the micro-benchmarking results, we noted that random access bandwidths on local memory tend to be slower than network bandwidths. We also consistently observed MPICH to perform better than OpenMPI, although neither managed to achieve network transfers of speeds exceeding 60MB/s. We have also proposed two designs of MPI based Global Scatter operation tailored towards facilitating spare-graph oriented irregular memory accesses across the cluster, viz. One-Sided Communication based and Two-Sided Communication based.

We found the Two-Sided implementation of global scatter operation to be 2x faster than the One-Sided variant on our ZedWulf cluster. Nevertheless, we still believe that the latter variant would benefit in RDMA ready systems. In comparison to naively executing edge updates using fine grained MPI transfers, our global scatter design yielded a speedup of 60x across varying number of MPI processes. We also formulated equations for predicting the overall time required for the global scatter operation and found it to have a coefficient of determination ($R^2$) of 99% while comparing one-sided scatter run-time measurements against the ones predicted by our model.

Running neural simulations based on our global scatter routine, we found the ZedWulf cluster to deliver the highest performance of 94 MTEPS across the tested x86 systems. With an energy efficiency of 0.7 MTEPS/Watt, our cluster surpassed that of Intel i7-4770 (best amongst tested) by over 38%.

VII. FUTURE WORK

While the implemented global scatter design is efficient, the underlying Kernel-Driver interaction likely limits the network throughput on the Zedboard. We plan to look into setting up Myrinet’s Open-MX message passing stack on all the nodes as it has been shown to improve performance for a wide variety of setups [13]. We also plan on tracing into MPI APIs to investigate the source of performance bottlenecks in the ARM variant with respect to x86. Advanced MPI-related studies can potentially scrutinize the MPI implementation for ARM to boost performance of frequently used APIs.

Further, as generation of graph data structures demands significant amounts of RAM, the limited DRAM resources and the inability of swap file to provide ample performance has restricted us from generating graphs beyond a few megabytes in size. Workaround using x86 machines as the graph generator needs to be explored and interfaced appropriately with our cluster. Also, we do not have a memory management unit (MMU) at the PL for managing irregular memory reads and writes originating from there. Thus, the PS has to explicitly exchange data with PL in an orderly manner, potentially resulting in excess overheads. Future studies should explore setting up a MMU in the PL such that the MMU can initiate and control data transfers between DRAM and PL.
Although this project focused on accelerating sparse graph problems, it lacked the infrastructure to actually integrate the FPGA resources along with MPI protocols to provide a fully functional graph machine. The next phase of this project will be to integrate existing Graph NoC (Network on Chip) FPGA designs on the PL and interface them to the PS using Xillybus drivers. In view of running High Performance LINPACK (HPL) and Graph 500 benchmarks, future works should also develop appropriate hardware implementations of kernels, such as BLAS, for leveraging the compute capabilities of the FPGA block.

REFERENCES

[21] X. LLC. Fpga coprocessing for c++ programmers (part v).
[29] S. Parker. Bg/q architecture.
[40] TG500. Top500.