HyperPRAW: Architecture-Aware Hypergraph Restreaming Partition to Improve Performance of Parallel Applications Running on High Performance Computing Systems

Carlos Fernandez Musoles
Daniel Coca
Paul Richmond
c.f.musoles@sheffield.ac.uk
d.coca@sheffield.ac.uk
p.richmond@sheffield.ac.uk
The University of Sheffield
Sheffield

ABSTRACT

High Performance Computing (HPC) demand is on the rise, particularly for large distributed computing. HPC systems have, by design, very heterogeneous architectures, both in computation and in communication bandwidth, resulting in wide variations in the cost of communications between compute units. If large distributed applications are to take full advantage of HPC, the physical communication capabilities must be taken into consideration when allocating workload. Hypergraphs are good at modelling total volume of communication in parallel and distributed applications. To the best of our knowledge, there are no hypergraph partitioning algorithms to date that are architecture-aware. We propose a novel restreaming hypergraph partitioning algorithm (*HyperPRAW*) that takes advantage of peer to peer physical bandwidth profiling data to improve distributed applications performance in HPC systems. Our results show that not only the quality of the partitions achieved by our algorithm is comparable with state-of-the-art multilevel partitioning, but that the runtime performance in a synthetic benchmark is significantly reduced in 10 hypergraph models tested, with speedup factors of up to 14x.

CCS CONCEPTS

• **Computing methodologies** → **Distributed algorithms**; • **Computer systems organization** → **Distributed architectures**; • **Theory of computation** → *Design and analysis of algorithms.*

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Carlos Fernandez Musoles, Daniel Coca, and Paul Richmond. 2019. HyperPRAW: Architecture-Aware Hypergraph Restreaming Partition to Improve Performance of Parallel Applications Running on High Performance Computing Systems . In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337876

1 INTRODUCTION

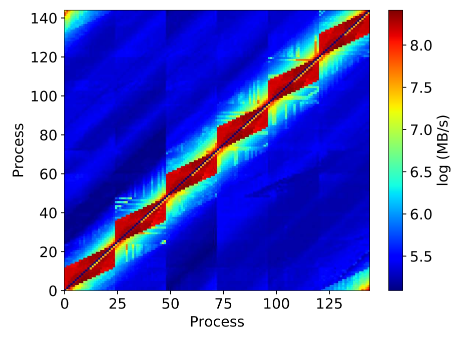
In the world of Big Data and large scientific simulations, there is huge demand for High Performance Computing (HPC) systems. HPC systems achieve high performance through parallelism and distribution. By the distributed nature of their architectures, there is a level of communication heterogeneity between any two processes within nodes. Take as an example the architecture of ARCHER[1], the UK National Supercomputer Service. Each compute node has two 12-core Intel Ivy Bridge processor. Four nodes are connected to an Aries router, 188 nodes are grouped into a cabinet; and two cabinets make up a group. There are all-to-all electric connections between nodes in the same group and all-to-all optical connections between different groups. This connectivity pattern, comparable to other HPC systems, leads to different connectivity speeds between computing units, depending on where they are hosted.

Figure 1A shows the profiled bandwidth (real communication speed) between any two computing units within a cluster of 6 nodes in ARCHER (144 units), indicating substantial differences between processes communication bandwidth depending on where the processes are hosted. The graph closely represents the architecture of the system, with the highest speed connectivity between computing units within the same processor (black), followed by communication between the two processors in the same node (dark grey). All other connectivity is considerably slower (light grey and white). In this example, all nodes belonged to the same group and we do not see any more intermediate bandwidths.

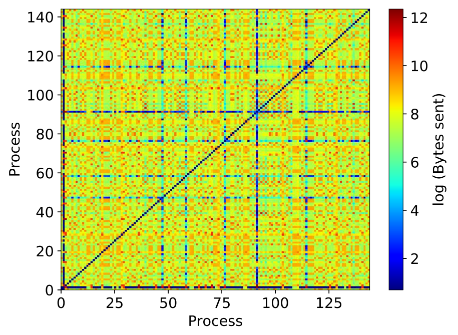
Data exchanged during a typical parallel application is shown in Figure 1B. The noisy pattern of the peer to peer data activity is a common feature of naive parallelism in which the total communication may be optimised (total data sent over the entire network) but only at a global level, not at individual unit to unit links.

The mismatch between the network bandwidth pattern and the actual data sent during simulation (Figure 1) leads to uneven costs of

[1]http://archer.ac.uk/



A



B

Figure 1: Discrepancies between network bandwidth in HPC systems and communication patterns in parallel applications. A: Peer to peer bandwidth heatmap on a 144 node job in ARCHER. B: Peer to peer communication activity pattern on a typical distributed application (run of our synthetic benchmark with *sparsine* hypergraph).

communication. Since bandwidths between units are significantly different, the cost of sending data (in terms of time) is also different.

Even though profile results have been shown only for ARCHER, the findings are generalisable to any HPC systems due to their distributed architecture. Parallel applications running in HPC systems can improve their communication performance and overall runtimes by considering the network bandwidth of the architecture they run on to reduce the real cost of communication.

2 RELATED WORK

Previous work has already highlighted the impact that uneven computation and communication architectures in HPC and Cloud computing has on computation performance [20, 21]. However, their domain problems were limited to graphs for graph processing. A hypergraph (a generalisation of graphs where edges can link n number of vertices) has been shown to model total volume of communication in parallel applications (first noticed by [4], also described in [9, 11]). Once the application is modelled as a hypergraph, partitioning algorithms can be used to optimise (reduce) the communication volume.

Partitioning algorithms for hypergraphs with good quality results exist using a variety of algorithms: multilevel partitioning (PaToH [5], hMetis [14], ParKway [18]) and multiconstraint [1, 9]. Unfortunately none of those approaches considers the physical architecture of the network. When modelling parallel applications as a hypergraph, not only it is important to reduce the hyperedge cut (connection between two vertices located in two different partitions), but also the hop cut (connections including the physical cost of communication).

Zoltan [10] offers a hierarchical approach for partitioning a hypergraph. It allows users to partition their hypergraphs at different levels of granularity, using a sequence of partitioning schemas (refinements on subgraphs). Each level can be used to model a level in the architecture hierarchy (socket, board, group, cluster, etc.). The focus of the approach is on being able to use high cost algorithms at levels where reducing communication is more important and low cost ones when the communication may not impact as much. However, this approach only establishes qualitatively differences between architecture levels and does not model well the cost of communication between computing units belonging to different hierarchies. This approach is not easily applicable in environments where the architecture is not know directly (in Cloud computing) or it is known but unreliable due to contextual circumstances (shared network resources).

To the best of our knowledge no previous work has focused on architecture-aware hypergraph partitioning. Previous attempts to architecture-aware graph partitioning exist and can be divided according to their partitioning strategy: local improvement or refinement with greedy strategies considering communication costs (PARAGON [22], kvMETIS [16], GrapH [15]); streaming greedy partitioning with communication cost as part of the allocation function ([20, 21]); and synchronous partitioning of the machine graph (model of the architecture) and the application graph (Surfer [8]).

A different strategy to optimise network communication in parallel applications is offered by the library *LibTopoMap* [13]. *LibTopoMap* maps MPI processes to arbitrary network topologies to direct high communicating processes to high bandwidth links. Note that this strategy does not redistribute work to minimise communication but rather maps the existing communication pattern on an application to a network architecture.

Parallel applications may not have constant communication patterns across their runtime. Using static approaches ahead of execution to distribute workload may not yield the best results in those circumstances. Repartitioning algorithms (those that perform the partitioning more than once) consider this and previous work proposes to model the cost of migrating data [6, 7] as part of the partitioning, in addition to cut minimisation. However streaming and restreaming approaches (see section 4) may be more suitable in large scale partitioning (frequently faster to execute as they work with local information only, less memory requirements and dynamic in nature), which is the case in problems such as large neuronal simulations or multiplication of very large sparse matrices. To the best of our knowledge, there are no streaming partitioners for hypergraphs that are architecture-aware.

3 KEY CONTRIBUTIONS

The goal of this work is to optimise distributed communication in HPC systems by reducing the mismatch observed in Figure 1. We propose a novel restreaming hypergraph partitioning that is architecture-aware to reduce the real cost of communication in distributed applications. This work makes the following contributions:

(1) Novel restreaming partitioning algorithm (*HyperPRAW*) for hypergraphs to reduce communication overhead on distributed applications compared to multilevel partitioning
(2) Using architecture information (physical bandwidth between computing units) further improves runtime performance by reducing real communication costs
(3) Refinement after reaching workload imbalance tolerance improves the quality of the partitioning

4 PROPOSED SOLUTION

We propose a restreaming algorithm to partition hypergraphs that incorporates information about the underlying architecture in which the modelled application will run.

Let us first define the problem. A hypergraph $H = (V, E)$ consists of a set of vertices V and a set of hyperedges E, where each hyperedge is a subset of V that defines the connectivity pattern. The size of each hyperedge is denoted as its cardinality. Hypergraphs are a generalisation of graphs that can have any cardinality, i.e., one hyperedge connects multiple vertices, where graphs have a maximum cardinality of 2. Hypergraph partitioning is a process that assigns vertices to partitions in such a way that a connectivity metric (usually hyperedge cuts, or hyperedges that span more than one partition) is minimised. To avoid trivial solutions that minimise the hyperedge cut (such as assigning all vertices to one partition) partitioning algorithms maintain load balancing by only allowing solutions that have a total imbalance factor that is below a specified value. The total imbalance is calculated dividing the maximum imbalanced partition in the scheme by the average imbalance across partitions. Formally:

$$\frac{\max_{p \in P}(L(p))}{(\sum_{i=0}^{|P|} L(p_i))/|P|}$$

where P is the set of partitions and $L(p)$ is the load cost for partition p defined as the sum of the weights of all its nodes, $L(p) = \sum_{i=0}^{N} W(n_i)$ where N is the number of nodes in partition p and $n_i \in p$. The total imbalance must be lower or equal than an arbitrary tolerance value.

Hypergraphs are good at modelling parallel communication when each hyperedge represents a frequent communication group of vertices. The more a hyperedge is cut (more partitions are involved) the more the modelled application will have to send data across partitions and hence more communication is required. Hypergraphs have been used in the past to model large scale distributed scientific simulations [12]. When using hypergraphs to model parallel applications, the goal is to partition the hypergraph in k partitions, where each partition represents a computing unit in the hardware architecture the application runs on.

Streaming graph partitioners differ from static comprehensive ones (such as k-way partitioning or recursive bisection) in that vertex allocation decisions are made based on local, partial information. This means the algorithm does not have the entire graph in view when calculating the cost of allocating a vertex to a partition. They are frequently called greedy since once they make a decision, it is not revoked later on after seeing more vertices. Figure 2 depicts the streaming process for the case of hypergraph partitioning. Figure 2A represents the input hypergraph, whereas Figure 2B models the current workload of each of the partitions based on current allocations. Figure 2C represents the streaming process, where one vertex is considered at a time. Based on local information, a value function is calculated per partition and the vertex is assigned to the one with higher value.

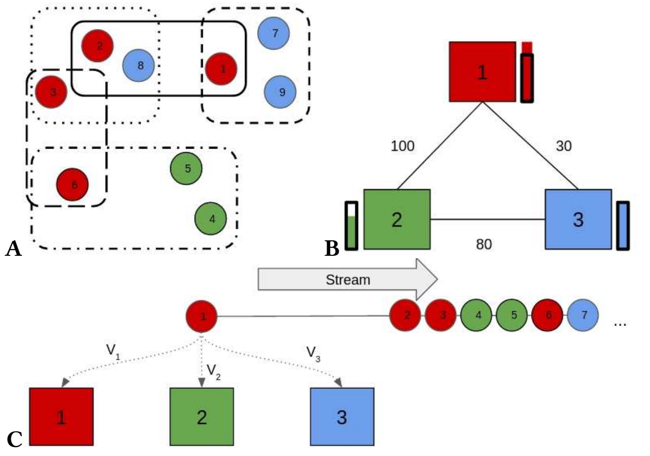


Figure 2: Overview of a hypergraph streaming partitioning process. A: Hypergraph, with vertices coloured based on the partition they are currently assigned to and hyperedges represented by dotted rectangles. B: model of the architecture and the current workload allocation; each box represents a partition (a compute unit in the architecture), with links weighed based on the physical peer to peer bandwidth of the architecture; an adjacent vertical bar represents whether the partition is currently overloaded or underloaded (measure of imbalance). C: Streaming process in which one vertex at a time is considered and assigned to a partition based on local information.

When an algorithm applies more than one pass (repeats the stream that visits the vertices once), it is often referred to as a restreaming approach. Our algorithm takes a similar approach to the graph restreaming software GRaSP [3] but it is applied to hypergraphs. To keep a good balance between the two opposing goals (workload balance and minimisation of total communication) we use a tempering parameter α that weighs the importance of workload imbalance. In the streaming partitioning algorithm FENNEL, [19] suggest a good starting value for α, which starts low:

$$\alpha = \sqrt(p) \times \frac{|E|}{\sqrt{|V|}},$$

where p is the number of partitions, $|E|$ is the number of hyperedges and $|V|$ is the number of vertices. After each stream this value is increased (the update parameter is set to 1.7). Our approach differs from GRaSP in two ways: the restreaming is allowed to continue until the partition is no longer improved (what we call the refinement phase) instead of stopping once the imbalance tolerance is reached; we reverse the tempering of the workload imbalance weigh once we are within imbalance tolerance—see section 6.1.

These two key differences allow for a refinement of the quality of the partition after reaching workload imbalance tolerance.

The *HyperPRAW* algorithm is described in Algorithm 1. The key step is the assignment of a vertex to a partition based on a value function. The partition that ends up with the highest value is the one to which the algorithm assigns the vertex. The next section describes what goes into calculating the value of assigning any vertex to each partition.

Algorithm 1: The *HyperPRAW* restreaming algorithm

```
Input  : $p$ (number of partitions); $\alpha$ (starting workload
         balance weight); $t_\alpha$ (workload balance tempering
         parameter); $imbalance\_tolerance$ (maximum
         imbalance tolerance); $N$ (maximum iterations)
Output : $P_k$, for $k = 1, ..., p$, where $P_k$ is the subset of $V$ that
         is allocated to partition $k$
Data   : $H = (V, E)$, where $V$ is a set of vertices and $E$ is a set
         of hyperedges for hypergraph $H$
Initialise $P_k$: Round robin assignment of each $v \in V$
Calculate $W(k)$ for each partition $k$
for $n = 1$ to $N$ do
    for $v \in V$ do
        $j \leftarrow \arg\max_{k=1,...,p} -N_k(v) \times T_k(v) - \alpha \frac{W(k)}{E(k)}$
        Add $v$ to set $P_j$
        Recalculate $W(j)$
    $\alpha \leftarrow \alpha \times t_\alpha$
    if $imbalance > imbalance\_tolerance$ then
        continue
    else if $Cost\ of\ P_k^n > Cost\ of\ P_k^{n-1}$ then
        return $P_k^{n-1}$
    else
        continue
return $P_k^N$
```

The algorithm has a computational complexity that grows with the number of iterations (N), the number of vertices ($|V|$) and hyperedges ($|E|$), the hyperedge cardinality and the number of partitions (p). The implementation of HyperPRAW can be found on the GitHub repository at https://github.com/cfmusoles/hyperPraw

4.1 Vertex assignment cost function

The value function $V_i(v)$ in equation 1 determines the value associated with assigning vertex v to partition i.

$$V_i(v) = -N_i(v) \times T_i(v) - \alpha \frac{W(i)}{E(i)} \qquad (1)$$

where $N_i(v)$ represents the number of partitions in which vertex v has neighbouring vertices (described by equation 2), $T_i(v)$ is the total cost of communication due to assigning vertex v to partition i (described in equation 4), $W(i)$ is the current workload of partition i and $E(i)$ is the expected workload for partition i. The parameter α weighs the importance of workload balance in the overall cost. Since it starts at a low value, the initial streams partition mostly based on communication cost. At later streams, the workload balance gains importance to achieve balanced partitions.

Throughout our experiments we have assumed even cost of computation per vertex and homogeneous work capacity for partitions, hence assigning one vertex to one partition increases by 1 its workload, and the expected workload for all partitions is the total workload divided the number of partitions. However the algorithm can easily account for heterogeneous computation and work capacities.

$$N_i(v) = \frac{\sum\limits_{j=0}^{p} A_j(v)}{p} \qquad (2)$$

$$A_j(v) = \begin{cases} 1, & \text{if } X_j(v) > 1 \\ 0, & \text{otherwise} \end{cases} \qquad (3)$$

The function $N_i(v)$ indicates the number of neighbouring partitions of vertex v, should it be placed in partition i. That is, in how many other partitions v has connecting vertices ($A_j(v)$ indicates whether vertex v has neighbours in partition j).

$$T_i(v) = \sum_{j=0}^{p} X_j(v) \times C(i, j) \qquad (4)$$

The function $T_i(v)$ computes the cost of communication associated with allocating vertex v to partition i. That cost, for another partition j is the number of neighbours v has in j ($X_j(v)$) multiplied by the cost of communication between partitions i and j ($C(i, j)$ which is discussed in section 4.2). The total communication cost is the sum of all costs for each partition other than the one v is assigned to (i).

In order to successfully calculate the total cost due to communication, the algorithm requires information regarding the cost of communicating between partitions (i.e., computing units). The next section describes how we map this cost.

4.2 Mapping cost of communication

The cost of communication matrix is derived from the peer to peer bandwidth calculated through profiling before HyperPRAW starts the streaming process. Discovery through profiling gives *HyperPRAW* flexibility as it can be applied to any architecture topology and it will discover the network costs automatically. This is an advantage in environments where the architecture is not known (in Cloud computing), or when it is known but unreliable due to contextual circumstances (shared network resources). If the bandwidth or communication costs are known, they can be used directly without the need for profiling.

When the cost of communication is done through profiling, first the bandwidth matrix is found, then it is transformed to represent cost of communication and not speed. The bandwidth profiling is done by iteratively sending data to and from MPI processes arranged in a ring formation[2] and timing how long it takes for them to reach back. One MPI process per computing unit is used.

With the peer to peer bandwidth data we calculate the cost of communication in the following way:

[2]The tool available in https://github.com/LLNL/mpiGraph is used for profiling bandwidth

$$C(i,j) = 2 - \frac{b_{ij} - b_{min}}{b_{max} - b_{min}}$$

where i and j represent two cores, b_{ij} is the bandwidth between core i and core j, and b_{min} and b_{max} are the maximum and minimum bandwidth between any two cores in the network. This normalises the costs to 1 for the fastest link, and 2 for the slowest. When $i == j$, $C(i,j) = 0$. The normalisation step helps *HyperPRAW* to be independent of the magnitude of bandwidth values. Since different hardware architectures can have different orders of magnitude bandwidths, the magnitude affects the balance between workload and communication cost used in the vertex assignment function (equation 1), potentially resulting in slower performance (if the cost values are too high) or sub-optimal solutions (if the cost values are too low, the stream can end underestimating the communication cost)

To accurately model the underlying architecture, the cost matrix must be calculated every time a new allocation of computing nodes is presented. In typical HPC jobs, this requires us to profile the architecture of the allocated cluster of nodes each time a job is started (since potentially new nodes are given).

4.3 Metric monitored during refinement: partitioning communication cost

To improve the quality of partitioning, we propose a refinement phase to the restreaming algorithm after the workload imbalance has reached values below the desired imbalance tolerance. During the refinement phase, the restreaming continues (i.e., further iterations are run) until a monitored quality metric ceases to improve. The metric selected is the partitioning communication cost. For a partitioning P, the partitioning communication cost $PC(P)$ is:

$$PC(P) = \sum_{i=0}^{k} \sum_{v} T_i(v), \text{ for all } v \in P_i \tag{5}$$

This uses the cost of communication $T_i(v)$ in equation 4 for all vertices and any partition i and aggregates it. Intuitively, this metric measures both the number of neighbours per vertex that live in different partitions to the vertex and the cost of communication between those partitions. This closely represents the volume and cost of communication in parallel applications that can be modelled with a hypergraph.

5 EXPERIMENTAL EVALUATION

5.1 Experimental design

To evaluate the performance of *HyperPRAW*, we use a public dataset[3] [17]. It includes a wide collection of hypergraphs used in various competitions (routability placement, circuit benchmark, SAT competition) and sparse matrices repositories. To test our approach we have selected 10 instances from within this collection that range in size, average cardinality and ratio number of hyperedge/vertex—see table 1.

We run all experiments in ARCHER, the UK National Supercomputing system. To ensure there is enough architecture heterogeneity, the job size is set to 576 cores. With 24 computing units per node this ensures we are using 24 different nodes in 6 blades.

Three experiments are carried out: the impact of refinement in restreaming (in section 6.1), partitioning quality evaluatio (in section 6.2), and runtime performance of hypergraphs on a synthetic benchmark (in section 6.3).

For both quality and runtime experiments we use a state-of-the-art multilevel recursive bisection partitioning algorithm (*Zoltan* implementation [10]) as a benchmark. To understand the impact of using the physical architecture cost of communication in our restreaming approach we use two versions of the algorithm: *HyperPRAW-basic* (where uniform cost of communication matrix is used) and *HyperPRAW-aware* (where cost of communication matrix from bandwidth profiling is used).

5.2 Quality and runtime metrics

Hypergraph partitioning algorithms traditionally optimise one of the two metrics: hyperedges cut (number of hyperedges that contain vertices that are allocated to more than one partition); Sum Of External Degrees (for each partition, sum of the hyperedges that are incident on the partition but not fully contained in it).

Formally, the Sum Of External Degrees (SOED) is $\sum_{i=0}^{k} |E(P_i)|$ for k partitions, where $E(P_i)$ is the number of hyperedges that are incident but not fully inside partition i. Intuitively, high values of SOED indicate hypergraphs are being cut across several partitions, representing more volume of communication.

Both hyperedge cut and SOED are cut-based metrics (calculated on the basis of hyperedges cut across partitions) and give an indication of the static quality of the partition. They are used in this work to report the quality of *HyperPRAW*. In addition, the partitioning communication cost defined in equation 5 is also used as a metric that combines cut information and physical cost of communication.

Quality of hypergraph partition only describes the results on the hypergraph itself. The hypergraph in this work is used as a model for a parallel application to improve performance. To measure the improvement that *HyperPRAW* can bring to these parallel applications, we use time execution on a synthetic benchmark—see section 5.3.

5.3 Synthetic runtime benchmark

Hypergraph partitioning is used in domains such as VLSI circuit design and boolean satisfiability problems. However those are static problems, that is once a solution has been found, there is no further problem. Hypergraphs can also model dynamic parallel applications to reduce total volume of communication in scientific simulations [12] and in sparse matrix multiplication [2]. In these cases, a good partitioning results in runtime improvements of the applications.

To measure the impact that our proposed strategy has on runtime communication, we design a synthetic benchmark. The benchmark is a null-compute simulation based on the input hypergraph and using a vertex allocation determined by the partitioning strategy selected. Since the simulation does not have any compute, it is purely communication-bound. The communication is proportional to hyperedge cut and SOED of the hypergraph and it is generated as follows: for each hyperedge on a given hypergraph, a message is sent to and from each vertex in the hyperedge if the vertices are

[3]Dataset is accessible via Zenodo at https://zenodo.org/record/291466

Table 1: Hypergraphs used in this work

Hypergraph	Vertices	Hyperedges	Total NNZ	Avg cardinality	hyperedge/vertex
sat14 itox vc1130 dual	441729	152256	1143974	7.51	0.34
2cubes sphere	101492	101492	1647264	16.23	1.00
ABACUS shell hd	23412	23412	218484	9.33	1.00
sparsine	50000	50000	1548988	30.98	1.00
pdb1HYS	36417	36417	4344765	119.31	1.00
sat14 10pipe q0 k primal	77639	2082017	6164595	2.96	26.82
sat14 E02F22	27148	1301188	11462079	8.81	47.93
webbase-1M	1000005	1000005	3105536	3.11	1.00
ship 001	34920	34920	4644230	133	1.00
sat14 atco enc1 opt1 05 21 dual	561784	59517	2167217	36.41	0.11

located in different partitions (computing units). This is repeated for all hyperedges in the hypergraph.

Although the synthetic benchmark is an extreme case of parallel application (no compute and communication of all connected compute elements on every time step) it adequately models a communication-bound distributed application (such as certain Spiking Neuronal Network simulations [12]). It allows us to compare the impact of different partitioning algorithms and understand how they improve communication in communication-bound distributed applications, in which scaling is limited by the overheads imposed with increased communication.

To account for variable network traffic and different nodes configurations provided by the job scheduler, the runtime experiments are run on three different jobs (hence different node placement and communication costs), with each job doing two iterations. Therefore the total number of simulations run per experiment is 6.

6 RESULTS

6.1 Refinement phase

To understand the effect of the refining phase, we compare the partition history of *HyperPRAW* for alternative stopping conditions of the restreaming process: no refinement (stop restreaming when the imbalance tolerance has been reached), refinement 1.0 and refinement 0.95 (continue restreaming until the partitioning quality is no longer improved). The number in the refinement alternatives refers to the update of the tempering parameter used once the imbalance tolerance is reached (1.0 results in the α parameter not being updated, whereas 0.95 decreases the value of α, and hence the importance of workload imbalance, instead of increasing it).

Figure 3 shows the partitioning history for 4 hypergraphs. The figure demonstrates how partitioning communication cost decreases with more iterations; this is the metric that is used to monitor and stop refinement and directly correlates with the amount of communication modelled by the hypergraph. Comparatively, both refinement strategies perform better than not refining at all. Using an update value for the tempering parameter that decreases the importance of workload balance (by a factor of 0.95 at each iteration) reaches the lowest levels of partitioning communication cost, therefore improving the restreaming quality.

6.2 Quality of partitioning

The quality of the partitioning of both versions of *HyperPRAW* and *Zoltan* is shown in Figure 4. In terms of standard hyperedge cut, *HyperPRAW* shows results that are below but comparable to *Zoltan* (from the 10 hypergraphs, the hyperedge cut is worse in 4, better in 2 and about the same in the other 4). When measuring the SOED, a metric that better models total volume of communication in parallel applications, the results are slightly better for *HyperPRAW* (3 worse instances, 1 about the same and 6 where it is better).

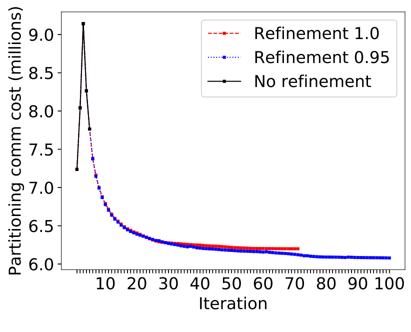
Neither the SOED nor the hyperedge cut include the physical cost of communication. The partitioning communication cost metric considers it and it is where *HyperPRAW* obtains the best results, with both versions versions improving over *Zoltan* on all hypergraphs and *HyperPRAW-aware* outperforming the basic alternative. Note that *HyperPRAW-aware* is the only one that uses the cost of communication matrix during the partitioning. Both *Zoltan* and *HyperPRAW-basic* assume uniform costs and only use the physical cost of communication to calculate the final partitioning cost.

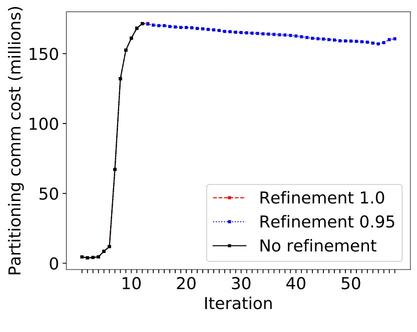
6.3 Runtime performance on benchmark

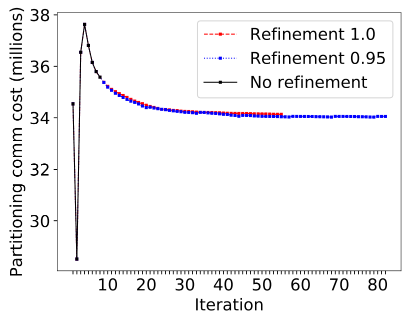
Figure 5 shows the overall runtime for 10 hypergraphs on our synthetic benchmark. The results show that *HyperPRAW-basic* reduces the simulation runtime with respect to *Zoltan* and *HyperPRAW-aware* further improves that significantly. The speed up factors of *HyperPRAW-aware* over *Zoltan* range from 1.3x to 14x.

7 DISCUSSION

In other restreaming partitioning algorithms, the iterative streams are halted when the workload imbalance tolerance is reached [3]. Nonetheless it is possible that the partitioning quality could improve if streams are allowed to continue despite being within acceptable imbalance. Figure 3 demonstrates the effectiveness of a refinement phase, where the streams continue until a partitioning metric ceases to be improved. Although the restreaming goes for longer (more iterations), this results in a higher quality partition (as measured by the total partitioning communication cost, a suitable metric for hypergraphs that model parallel communication). The best alternative is found to be when during the refinement phase, the workload imbalance weight parameter α is reduced (refinement 0.95), instead of increased as it is done when outside of workload imbalance tolerance (update value of 1.7). The value 0.95 was found experimentally,







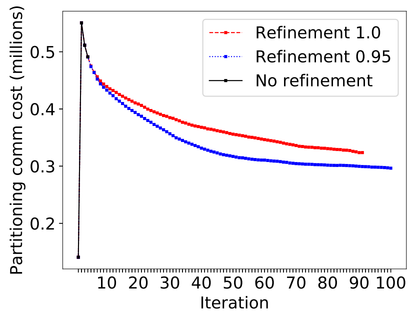


Figure 3: Partition history of the *HyperPRAW* algorithm comparing different refinement strategies: no refinement (black), refinement 1.0 (red dashed) and refinement 0.95 (blue dotted). A: *2cubes sphere* hypergraph. B: *sat14 itox vc1130 dual* hypergraph. C: *sparsine* hypergraph. D: *ABACUS shell hd* hypergraph.

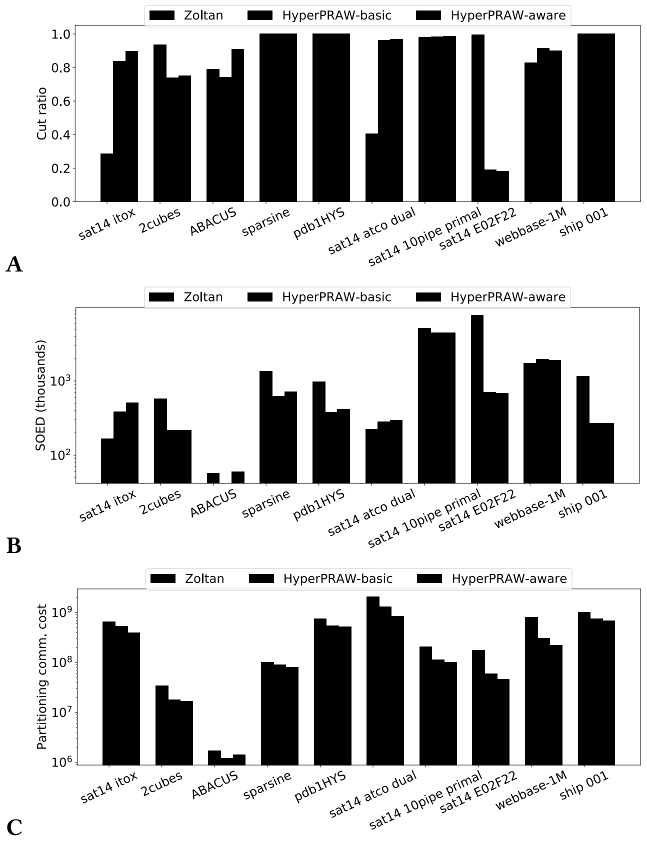


Figure 4: Quality metrics on 10 hypergraphs comparing the partitioning algorithms: *Zoltan* (black), *HyperPRAW-basic* (orange vertical lines) and *HyperPRAW-aware* (yellow horizontal lines). A: Hyperedge cut. B: Sum of External Degrees (SOED) in logarithmic scale. C: Partitioning communication cost in logarithmic scale.

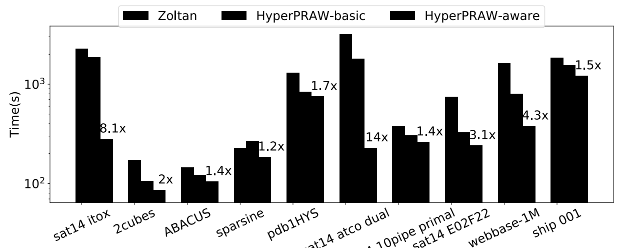


Figure 5: Runtime performance (in logarithmic scale) on a synthetic benchmark for 10 hypergraphs comparing the partitioning algorithms: *Zoltan* (black), *HyperPRAW-basic* (orange vertical lines) and *HyperPRAW-aware* (yellow horizontal lines). The speedup factors of *HyperPRAW-aware* over *Zoltan* are annotated in the figure.

since lower values may force the algorithm to fluctuate in and out of the acceptable load balance range, degrading performance. Nonetheless, the intuition is that the best partition is found when the balance constraints are relaxed (an extreme case is when all workload is assigned to one partition; there is no cut or communication in such an assignment, but we have maximum imbalance). Once the algorithm is within the range of acceptable partitions (i.e., within the imbalance tolerance), we can search for slightly more imbalanced solutions in an attempt to find an acceptable solution that is maximally imbalanced.

Figure 3 also shows that the effectiveness of the refinement phase varies with the hypergraph, indicating that some types of hypergraphs do benefit more from a refinement than others. The refinement factor is therefore a candidate parameter to tune empirically.

Figures 4A and 4B show the cut-based metrics. In hyperedge cut, *HyperPRAW* underperforms *Zoltan* in 4 of the 10 hypergraphs, but in SOED the *Zoltan* benchmark is outperformed in 6 hypergraphs, resulting in an overall comparable performance. When physical communication cost matrix is considered, as is the case with partitioning communication cost (Figure 4C), the quality metric shows *HyperPRAW* being consistently superior than *Zoltan* in all hypergraphs. This work attempts to improve the performance of distributed applications, for which we model the application as a hypergraph that is partitioned using our proposed restreaming approach. The output from the restreaming algorithm is a scheme that is used to distribute workload in a heterogeneous environment and its quality is ultimately evaluated indirectly by timing how long the application runs for under the new scheme. Therefore our focus is on end runtime performance improvement, and although we report traditional partitioning quality metrics, this is done descriptively and not as a target metric. This is the reason to report the partition quality with the partitioning communication cost, since it reflects more accurately the needs of the target to optimise (peer to peer communication in a heterogeneous environment). Therefore, the resulting quality of the restreaming partitioning shown in Figure 4C indicates a net improvement over *Zoltan*.

The runtime performance on the synthetic benchmark in Figure 5 confirms the results obtained in the partitioning communication cost quality metric. In 9 hypergraphs, both versions of *HyperPRAW* outperform *Zoltan*, showing the effectiveness of the proposed restreaming approach. The results also show that the restreaming approach benefits from using the physical communication cost matrix, where in all cases *HyperPRAW-aware* achieves faster simulation times than the basic counterpart and *Zoltan*. When compared with *Zoltan*, *HyperPRAW-aware* reaches significant speedup factors ranging from 1.3x to 14x (with 3 hypergraphs reaching speedups above 4x).

A paradigmatic case of the importance of the partitioning communication cost metric over the cut-based ones is found on two hypergraphs: *sat14 itox vc1130 dual* and *sat14 atco enc1 opt1*. In both cases, the hyperedge cut and the SOED metrics are worse on the restreaming approach than in *Zoltan*. However, the partitioning communication cost is better in *HyperPRAW*, with an outstanding runtime speedup on the synthetic benchmark of 8.1x and 14x respectively.

HyperPRAW relies on the information built through profiling to construct the communication cost matrix. Using a simple MPI send-receive ring protocol is empirically seen to be sufficient to successfully map the known hardware topology in ARCHER. Figures 1A and 6A show the characteristic 24 process clusters of high speed communication in ARCHER, which map to cores within a single computing node. Within a 24 process cluster we also see two tiers, corresponding to the two 12 cores Intel Ivy Bridge processors.

Earlier we evidenced the issue of running parallel applications in heterogeneous HPC systems by profiling the communication pattern of distributed application and the peer to peer bandwidth of a group of computing units.. Figure 6A shows the peer to peer bandwidth for a job allocation in ARCHER where we run the synthetic benchmark. As expected from its architecture, the fastest communication links are on neighbouring units (each group of 24 units belonging to the same computing node), thus the pattern of high bandwidth in the central band. For an optimal utilisation of the hardware architecture, the patterns of activity of the parallel application should resemble that of the bandwidth profile. This is what we see when showing the pattern of activity of our synthetic benchmark for sparsine hypergraph partitioned with *Zoltan* (6B), *HyperPRAW-basic* (6C) and *HyperPRAW-aware* (6D). For the first two, since they do not use the physical communication cost matrix, the pattern of communication is uniformly random. However, for *HyperPRAW-aware*, using the communication cost matrix makes the restreaming distribute the communication pattern to closely resemble the peer to peer bandwidth. Therefore our approach is able to better exploit fast interconnections between computing units.

8 CONCLUSION AND FURTHER WORK

8.1 Conclusion

This work demonstrates the importance of being architecture-aware when distributing workload in HPC systems in parallel applications. We propose *HyperPRAW*, an architecture-aware restreaming partitioning algorithm that optimises communication by understanding the underlying network bandwidth. In conclusion:

(1) architecture-aware restreaming algorithm increases parallel application performance on a synthetic benchmark up to 14x compared to multilevel recursive bisection.
(2) *HyperPRAW* is able to better exploit fast interconnections between computing units, which make a small percentage of the total interconnections on HPC systems.
(3) Refinement after reaching workload imbalance tolerance improves the quality of the partitioning.

8.2 Further work

The current work shows there is scope to improve the performance of parallel applications in HPC systems, where architecture heterogeneity is unavoidable. One limitation of this work is that the restreaming partitioning is performed sequentially on a single processor. This limits the applicability on larger scales (high number of partitions and larger hypergraphs). This limitation can be removed if the restreaming algorithm were to be adapted to parallel execution. Battaglino et al. [3] demonstrate that parallel streaming with minimal quality loss is possible by creating one stream per

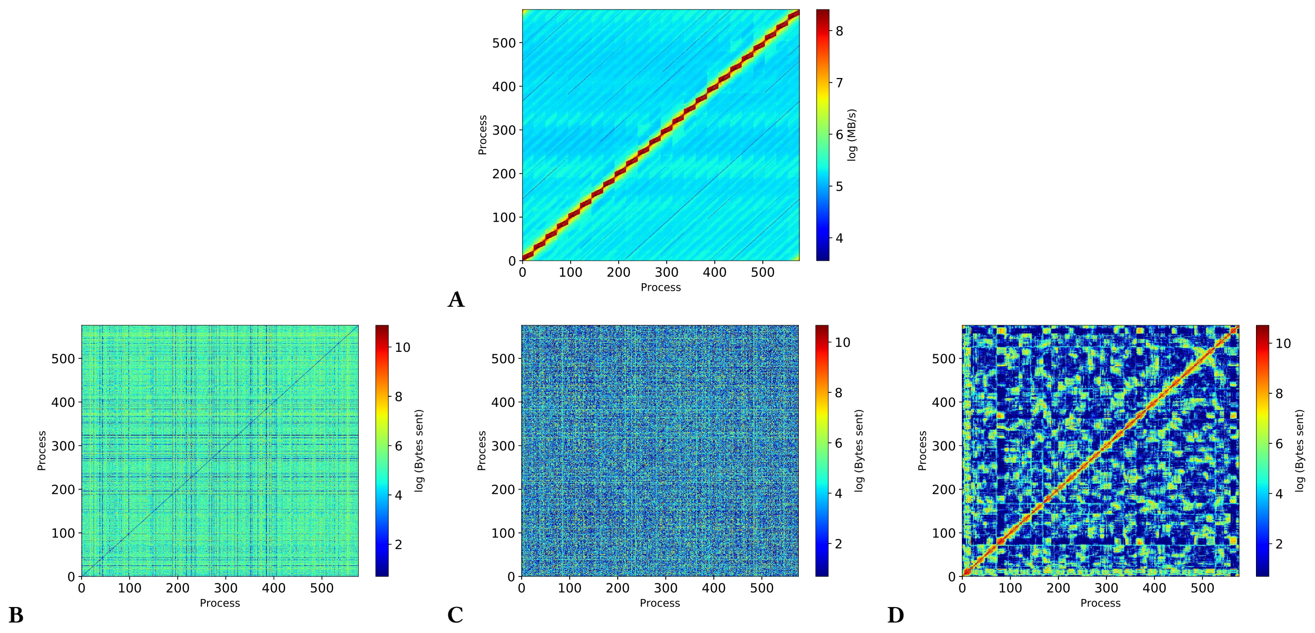


Figure 6: Architecture bandwidth compared to peer to peer communication pattern on the synthetic benchmark. A: peer to peer bandwidth of a 576 computing units job in ARCHER. The bottom part of the figure represents the peer to peer communication pattern on the synthetic benchmark run of the sparsine hypergraph: B: communication pattern after using *Zoltan*; C: Communication pattern using *HyperPRAW-basic*; D: Communication pattern using *HyperPRAW-aware*.

partition and periodically synchronising workload and partition assignments. This is identified as an area of future work.

We have assumed that the communication costs remain constant throughout the lifespan of the distributed application. If the costs vary (for instance due to network contention or other competition for resources in HPC systems) then the effectiveness of the strategy may diminish. In scenarios where this contention is likely to impact performance, it may jsutify doing dynamic profiling to update the communication cost.

We have proven the applicability of the restreaming approach on a synthetic benchmark. Future work should attempt to apply *HyperPRAW* on applications which have the potential to benefit from better communication distribution in large systems. Two identified domains are the simulation of Spiking Neuronal Networks and sparse matrix multiplications. Both of them have been modelled as hypergraphs to improve parallel performance ([2, 12]) and *HyperPRAW* could further improve it by optimising communication.

For simplicity, we do not attempt to model dynamic communication patterns or asymmetric communication patterns (where some hyperedges may communicate more than others). Both can be tackled by weighing hyperedges and consider the cost of partitioning accordingly. This can be accommodated into *HyperPRAW* by weighing the cost of communications in the vertex assignment objective function with the hyperedge weight. This is an interesting area of future work that may impact performance in highly dynamic asymmetric parallel applications.

ACKNOWLEDGMENTS

The authors would like to thank the UK National Supercomputing Service for providing access to ARCHER cluster computer, enabling us to perform the necessary experimentation.

This work was supported by BBSRC "The Digital Fruit Fly Brain" under GrantNo.: BB/M025527/1, EPSRC "Accelerating Scientific Discovery with Accelerated Computing" under GrantNo.: EP/N018869/1 and access to ARCHER through an EPSRC Grant (project e582).

REFERENCES

[1] C. Aykanat, B. B. Cambazoglu, and B. Uçar. 2008. Multi-level direct K-way hypergraph partitioning with multiple constraints and fixed vertices. *J. Parallel and Distrib. Comput.* 68, 5 (2008), 609–625. https://doi.org/10.1016/j.jpdc.2007.09.006

[2] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz. 2016. Hypergraph Partitioning for Sparse Matrix-Matrix Multiplication. *arXiv:1603.05627v1 [cs.DC]* (2016). https://doi.org/10.1145/2755573.2755613

[3] C. Battaglino, P. Pienta, and R. Vuduc. 2015. GraSP: Distributed Streaming Graph Partitioning. *Proceedings of the 1st High Performance Graph Mining Workshop - HPGM '15* (2015). https://doi.org/10.5821/hpgm15.3

[4] U. V. Catalyurek and C. Aykanat. 1999. Hypergraph-Partitioning-Based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS* 10, 7 (1999), 673–693.

[5] U. V. Catalyurek and C. Aykanat. 1999. *PaToH: Partitioning Tool for Hypergraphs.* Technical Report.

[6] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. BozdaÇğ, R. Heaphy, and L. A. Riesen. 2007. Hypergraph-based dynamic load balancing for adaptive scientific computations. *Proceedings - 21st International Parallel and Distributed Processing Symposium, IPDPS 2007; Abstracts and CD-ROM* (2007), 1–11. https://doi.org/10.1109/IPDPS.2007.370258

[7] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. BozdaÇğ, R. T. Heaphy, and L. A. Riesen. 2009. A repartitioning hypergraph model for dynamic load balancing. *J. Parallel and Distrib. Comput.* 69, 8 (2009), 711–724. https://doi.org/10.1016/j.jpdc.2009.04.011

[8] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li. 2012. Improving large graph processing on partitioned graphs in the cloud. *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12* (2012), 1–13. https://doi.org/10.1145/2391229.2391232

[9] M. Deveci, K. Kaya, B. Ucar, and U. V. Catalyurek. 2015. Hypergraph partitioning for multiple communication cost metrics: Model and methods. *J. Parallel and Distrib. Comput.* 77 (2015), 69–83. https://doi.org/10.1016/j.jpdc.2014.12.002

[10] K. D. Devine, Erik G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. 2006. Parallel hypergraph partitioning for scientific computing. *20th International Parallel and Distributed Processing Symposium, IPDPS 2006* 2006 (2006), 1–15. https://doi.org/10.1109/IPDPS.2006.1639359

[11] K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Teresco, J. Faik, J. E. Flaherty, and L. G. Gervasio. 2005. New challenges in dynamic load balancing. *Applied Numerical Mathematics* 52, 2-3 SPEC. ISS. (2005), 133–152. https://doi.org/10.1016/j.apnum.2004.08.028

[12] C. Fernandez-Musoles, D. Coca, and P. Richmond. 2019. Communication Sparsity in Distributed Spiking Neural Network Simulations to Improve Scalability. *Frontiers in Neuroinformatics* April (2019), 1–15. https://doi.org/10.3389/fninf.2019.00019

[13] T. Hoefler and M. Snir. 2011. Generic topology mapping strategies for large-scale parallel architectures. *Proceedings of the international conference on Supercomputing - ICS '11* (2011), 75. https://doi.org/10.1145/1995896.1995909

[14] G. Karypis and V. Kumar. 1999. Multilevel k -way Hypergraph Partitioning. In *36th Design Automation Conference*, Vol. 11. New Orleans (USA), 285 – 300. https://doi.org/10.1145/309847.309954

[15] C Mayer, M. A. Tariq, C. Li, and K. Rothermel. 2016. GrapH: Heterogeneity-Aware Graph Computation with Adaptive Partitioning. In *2016 IEEE 36th International Conference on Distributed Computing Systems*. 118–128. https://doi.org/10.1109/ICDCS.2016.92

[16] I. Moulitsas and G. Karypis. 2008. Architecture aware partitioning algorithms. *International Conference on Algorithms and Architectures for Parallel Processing* 5022 (2008), 42–53. https://doi.org/10.1007/978-3-540-69501-1{_}6

[17] S. Schlag. 2017. A Benchmark Set for Multilevel Hypergraph Partitioning Algorithms [Data set].

[18] A. Trifunovic and W. Knottenbelt. 2008. Parallel Multilevel Algorithms for Multi-constraint Graph Partitioning. *J. Parallel and Distrib. Comput.* 68, 5 (2008), 563 âĂŞ 581. https://doi.org/10.1007/3-540-44520-x{_}39

[19] C. E. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic. 2012. *FENNEL : Streaming Graph Partitioning for Massive Scale Graphs Categories and Subject Descriptors*. Technical Report. https://doi.org/10.1145/2556195.2556213

[20] N. Xu, B. Cui, L. Chen, Z. Huang, and Y. Shao. 2015. Heterogeneous Environment Aware Streaming Graph Partitioning. *IEEE Transactions on Knowledge and Data Engineering* 27, 6 (2015), 1560–1572.

[21] J. Xue, Z. Yang, S. Hou, and Y. Dai. 2015. When computing meets heterogeneous cluster: Workload assignment in graph computation. *Proceedings - 2015 IEEE International Conference on Big Data* (2015), 154–163. https://doi.org/10.1109/BigData.2015.7363752

[22] A. Zheng, A. Labrinidis, P. Pisciuneri, P. K. Chrysanthis, and P. Givi. 2016. PARAGON: Parallel Architecture-Aware Graph Partition Refinement Algorithm. In *19th International Conference on Extending Database Technology (EDBT*. 365–376. https://doi.org/10.5441/002/edbt.2016.34