An effective scheme for memory congestion reduction in Multi-core environment

Navin Mani Upadhyay, Ravi Shankar Singh

PII: S1319-1578(20)30388-8
DOI: https://doi.org/10.1016/j.jksuci.2020.05.011
Reference: JKSUCI 796

To appear in: *Journal of King Saud University - Computer and Information Sciences*

Received Date: 4 February 2020
Accepted Date: 31 May 2020

An effective scheme for memory congestion reduction in Multi-core environment

Abstract

In modern NUMA systems, increasing number of cores lead to a heavy congestion on shared caches and memory controllers. Therefore, in this paper, we propose a method (RCLB) which is capable of reducing the memory congestion with reconcile locality of communication. It prevents the load imbalancing on memory controllers which identifies the data traffic applying through clusters using MPI communication patterns. The proposed method works at the running time of MPI application with a small modification in the profiling technique. We have tested the kernels of NAS parallel benchmarks and found it as state-of-art. The experimental results show that the proposed method has achieved a bandwidth improvement ranges 12.65-23% in memory-based tests (CLB).

Keywords: High-Performance Computing, MPI Process placement, Memory Congestion, NUMA Architecture, Multi-Cores, MPI-Routines

1. Introduction

The number of cores in a processor has been increased from the last few decades. Based on such cores placed on processors, one can easily classify the system, because all processor's cores are not necessarily connected in a sequential manner. There may be some processor's core available, which are embedded in a parallel fashion. They can share only one cache memory available for their group. In such architecture, the memory sharing problem may happen for several reasons. The most important reason is memory congestion and this is the biggest problem for HPC. The memory congestion will be increased if the scale of systems will increase [1]. The scaling of the systems is categorized into two types: strong and weak scaling. The strong scaling systems have parallel arrangements of communication between their processors, whereas in weak scaling systems the parallel processing can be a challenge. Therefore reducing the memory congestion from weak scaling systems is the main goal of this paper so that the performance will increase.

Moreover, such cores are very useful in High-Performance Computing and a large-scale multi-processing system environment. In such environment, each processor can be a part of one or more than one group of processors. Such groups of processor's cores are called a node and they are associated with common shared memory, memory controller and multiple interconnected links. One thing is notable that, all the memory controllers are logically shared by all processor's cores available in all the groups physically [2–5].

1.1. Memory congestion Problem

Memory congestion is concerned with the allocation of resources among all the connected nodes in a network. The connection will be as such that the nodes can operate the transaction at an acceptable performance level. In such a case, when the demand exceeds or it will work just near the capacity of the network resources. The network resources include link bandwidth, memory size, and processing capacity at all intermediary nodes. However, the resource allocation is necessary, even for a low load, but, the problem becomes challenging when the load increases. Due to a massive load on a single node, the fairness issues will occur, and low overhead will increase, which reduces the performance.

To make the problem more clear and understandable, figure 1 shows the proposed architecture with four nodes and twelve processors: (four processors each node) which is same as the problem taken by Mulya et al [6, 7]. Every node is connected with a local DRAM and a memory controller. All the nodes can access the data through memory controllers and interconnected links. These controllers are generally a high-speed connection such as QP Interconnect [8]. However, the nodes are connected with high-speed interconnections but the other connected node needs a higher latency than parental data node [4, 9–12]. The memory latency depends on the location of data, from where it is to be fetched. So, to express data movement explicitly, this work uses MPI for writing its proposed algorithms. Based on a

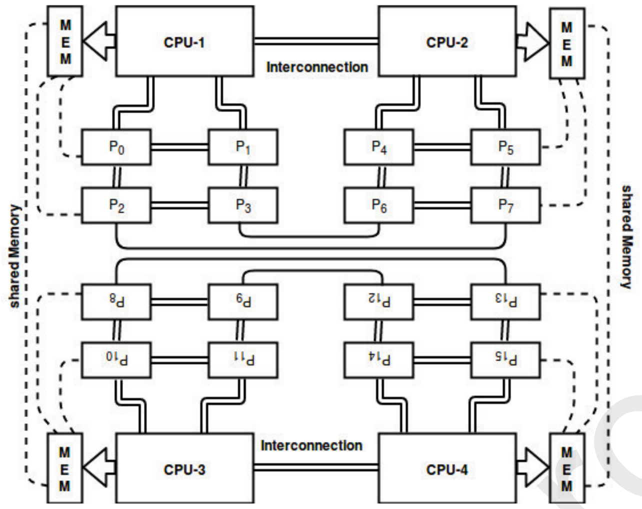


Figure 1: Connection-topology of used machines.

problem discussed in the paper [6, 7], this paper discusses about the process placement strategies available for different kernels: CG, FT, LU and MG [9, 13, 14]. The placement method is a crucial task for NUMA based systems [15], but it can be achieved by mapping techniques available in MPI processes with running processors [13]. Each mapping has a unique ID number called MPI-rank. Practically the process managers of MPI tags each MPI processes to a core according to their mapping. As a result, all the conventional work become capable to access the data locally to reduce the penalty occurred during execution.

The data accessing technique through locality is also improved by using the MPI placement strategies since the placement methods use matching of communication patterns. These communication patterns are mapped towards underlying hardware. If the number of processors core are not large at any node then the performance can be improved by decreasing the number of accessed communication patterns at memory controllers. A recent study has been presented by [2] to optimize the memory congestion at memory controllers. In this study, they have presented a significant increase in the performance of modern Systems. The demerit of their presented approach is a longer delay at the memory controller. In the modern NUMA systems, the delay is high since the access takes a longer traversing of physical distance to reach one to another node. During execution, if the number of cores in any node increases, then the memory congestion at any of the connected memory controllers may increase and the overall performance will decrease whereas the locality plays an important role in reducing the traffic on interconnected links. Therefore reconciling of locality and congestion becomes a challenging problem in modern NUMA systems [15]. In this paper, a method to address the memory congestion problem through MPI process placement technique has been proposed.

1.2. Motivation

The scientific computation based processes needs more shared memory, transaction from one to another node and a good management for large number of data collections collected from worldwide. The computations like weather forecasting, earthquake prediction, studies under deep sea area etc., need a high speed and scalability as an essential component where they combine a high-end computing technologies with a high-performance network and a wide-area storage management techniques. In such system, since they worked parallelly so, any process arrive at any node and may execute at several nodes before exit. This may happen only because of the dynamic nature of parallel computing system. Based on the demand some nodes add while some nodes stand in a ideal position. As a result, the waiting time at heavily loaded nodes increases and the performance of the system will degrade. This may happen because the memory queues at heavily loaded nodes becomes full, and further processing requests cannot be enqueued. This situation causes unavailability of memory problem, which usually occurs during the execution of such scientific calculation which requires a large data-processing. In this situation several processing requests have arrived in at a node in a short period of time. As a result, most of the computation cannot be completed within their respective deadlines. Therefore the balancing of task allocation will become important for reducing memory congestion in such NUMA systems.

So, this problem motivates us to address the memory congestion problem and work for its solution.

1.3. Contribution

The proposed method considers the place of congestion by monitoring the communication patterns, analyzes the communication traffic at interconnect links and then decide the correct mapping for the placed processes. These tasks are done in advance, so the load imbalance among processors is also become reduced. An MPI based algorithm called Reducing congestion with load balancing (RCLB) is proposed which determines the process mapping among nodes, reduces the memory congestion at the memory controllers and improves the performance of the system. This paper also compares the proposed method with some existing placement methods CG, MG, LU and FT [9, 13, 14]. The proposed method also includes a low-level on-line monitoring tool to capture the data, MPI communication patterns, data clustering and data traffic places. At last, for the validation purpose, performance evaluation through simulation and experiments has been conducted. The evaluation results show that the placement of processes calculated by the proposed method achieves a lesser execution time than two greedy policies (Packed and Socket-span) and one locality optimized method (Tree-match) [16].

1.4. Paper Organization

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 discuss about prerequisites (Notations, definitions and assumptions). Section 4 Explains about problem statement and formulation. Section 5 shows a detailed study about proposed algorithms. In section 6 step by step proposed algorithms has been explained. In section 7 the Experiments that validate the proposed method are analyzed and discussed. Finally, Section 8 gives the conclusions and the future guidelines.

2. RELATED WORK

This section presents some recent literature survey so that by reading this section, one can easily understand the level of work and find-out possibilities with work-done so far in this area. In this regard, we have divided our survey into three categories. First is based on MPI Process Placement, second is based on Load Balancing and third is based on Memory Placement on Modern NUMA Systems [15].

- **Load Balancing** : In this regards, several works have been done by many researchers. However, every proposed work is important and it is very difficult to include all the works here. So we are adding a few recent and advance work here in this paper. Bhatele et.al. [17] and Mulya et.al.[7] have presented the advantages of load balancing algorithms for molecular dynamics applications. One more author Pilla et.al. [18] have proposed a Hierarchical Approach for Load Balancing on Parallel Multi-core Systems, where they have found a significant result. For CPU load characterization, HwTopoLB [19], and Treematch LB [20] have computed latencies of network links in their proposed algorithm. They have used Charm++ and task migration technique. Another work in the field of the task placement has been done [21–23] in which they have found a good scheduler result to improve load balance. One more author Mercier [21] try to improve the placement of MPI processes by combining hardware's hierarchy information[24]. The Hwloc [5] has gathered some NUMA machines on which jeannot [25] has worked and proposed a hierarchical method to improve the performance of the NUMA system.

  The major differences between the aforementioned work and the proposed work is as follows:

  – The proposed work is able to balance the traffic on memory controllers by analyzing the communication patterns.

  – The processes are assigned at the time of their launching so there is no migration time required for processors core. Thus it does not depends on a specific runtime.

- **Process Placement** : Many researchers have performed several important works in related studies, which are being introduced through this paper. The MPI framework[26] is used to process different nodes of the cluster based on the performance profile. Many researchers have performed several important works in related studies, which are being introduced through this paper. The MPI framework[26] is used to process different nodes of the cluster based on the performance profile. A Graph-Based partitioning method has been proposed by Hendrickson et al. in which they have shown a significant result for optimizing process mapping. Another algorithm has been proposed by Zhang et al[27] and Ma et al [27]. They have found that most of the work relies on profiling techniques which are off-line. The trace of communication between processes and cores have also analyzed by them [6, 16, 26, 27]. One author, Jeannot et al [16] have proposed a recent work with a tree-match based algorithm, which is not so advance but similar to the chosen problem [7]. In this work, they have projected the communication patterns of NUMA architectures. A similar and advance method have proposed by Mahato et al and Iman et al [28, 29]. They have computed the process placement for load balancing and memory congestion. One more author Tan et al [15] have introduced a method with a comparative study of Tree-match technique. They have used a graph-based partitioning algorithm for tracing the communication pattern. The drawback of such methods are only the requirement of off-line profiling techniques. Off-line profiling techniques are high overhead and time-consuming in nature. One more method has been proposed on the basis of off-line profiling technique by Upadhyay et al [30]. The proposed method does not have these type of disadvantages because it maps dynamically at runtime. We have observed two key differences between the above-mentioned related work and the proposed work.

  – It does not employ a communication tracing mechanism since the communication patterns have been traced directly by MPI routines.

  – The related work mainly focuses on the data access mechanism: locality or remote-accessed.

  In general terms, we can say that this work focuses on MPI processes using placement methods based on NUMA systems. It maps the communication patterns directly and reduces memory congestion.

- **Memory Placement** : Some more work related to other architecture has been done by several researchers. Some of the similar work on such techniques based on CUDA have proposed by Iman et al [29]. The wrapper exposes API calls, minimizes the communication cost by synchronizing with torque batch systems, which overrides to some device management run on CUDA. The researcher has also proposed a topology-aware selection[28, 29] scheme for GPUs based on MPI processes to reduce memory congestion. By using the SCOTCH library and graph mapping algorithm, Iman et al [29] has proposed an off-line profiling technique for MPI libraries and applications. Another researcher Samuel Greengard [31], have also proposed his work to minimize the communication cost without considering the performance interference from scheduled jobs. On the basis of performance, several researchers have investigated about scheduling algorithms for low-level resource[32] allocation and minimizing[33] them along with CPU-based workload distribution [34–36] as-well-as GPU-based workload distribution [37, 38]. However, all these papers describe the performance bottlenecks for CPU applications. One author Dashti et al [39], proposed a memory placement algorithm which is different from all(mentioned earlier in this section). Their proposed algorithm is capable of reconciling the data location and reducing memory congestion. Only the key difference is Dashti et al [39] uses their algorithm as a Linux kernel policy for dynamically memory allocation to avoid congestion.

3. NOTATIONS, DEFINITIONS, ABBREVIATIONS AND ASSUMPTIONS

3.1. Basic definitions

- **NAS Parallel Benchmarks**[14]: The NAS parallel benchmarks are a group of programs designed as a part of the NASA Numerical Aerodynamic Simulation (NAS) program initially to evaluate supercomputers. They mimic the computation and data movement characteristics of large-scale computations. NAS parallel benchmark suite consists of five kernels (EP, MG, FT, CG, IS) and three pseudo applications (LU, SP, BT) programs. NPB implementation written in MPI and distributed by NASA.

- **Multi-Grid (MG) kernel**[9, 14]: MG is a simplified multigrid kernel. MG uses highly structured (both short and long-distance) communication. The default entries are the unmodified NPB kernel, hence utilizing more than one processor per node is achieved through multiple MPI processes on each node. The MG benchmark carries out computation at a series of levels.
- **Conjugate Gradient (CG) kernel**[9, 14]: In CG, a conjugate gradient method used to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is typical of unstructured grid computations in that it tests irregular long-distance communication, employing unstructured matrix-vector multiplication.
- **Fourier Transform (FT) kernel**[9, 14]: The FT kernel solves a 3-D partial differential equation solution using FFTs. This kernel performs the essence of many spectral codes. It is a rigorous test of all-to-all communication performance.
- **Lower-Upper (LU) Gauss-Seidel kernel**[9, 14]: The Lower-Upper (LU) Gauss-Seidel, Solves a synthetic system of nonlinear Partial Differential Equations (PDEs) using three different algorithms involving block tridiagonal, scalar Penta-diagonal and symmetric successive over-relaxation (SSOR) solver kernels, respectively.
- **Uncore**: Uncore is a term used by Intel to describe the functions of a microprocessor that are not in the core, but which must be closely connected to the core to achieve high performance. Uncore functions include QP Interconnection controllers, L3 cache, snoop agent pipeline, on-die memory controller, and Thunderbolt controller.
- **LLC**[40]: The LLC (last-level cache) is the last, and longest-latency, level in the memory hierarchy before main memory (DRAM). Any memory requests missing here must be serviced by local or remote DRAM, with significant latency. The LLC Miss metric shows a ratio of cycles with outstanding LLC misses to all cycles.
- **IPC**[41]: In computer architecture, instructions per cycle (IPC), commonly called Instructions per clock is one aspect of a processor's performance: the average number of instructions executed for each clock cycle. It is the multiplicative inverse of cycles per instruction.
- **Memory Latency**[9, 14, 41]: In computing, memory latency is the time (the latency) between initiating a request for a byte or word in memory until it is retrieved by a processor.
- **Memory Controller Imbalance**[42]: Memory controller imbalance is the standard deviation of the bandwidth utilization of the memory controllers across all nodes.
- **Goodness-Of-Fit**[6, 7, 9, 14]: The goodness of fit test is a statistical hypothesis test to see how well sample data fit a distribution from a population with a normal distribution. Put differently, this test shows if your sample data represents the data you would expect to find in the actual population or if it is somehow skewed. Goodness-of-fit establishes the discrepancy between the observed values and those that would be expected of the model in a normal distribution case.

3.2. Notation Used:

All the used symbols in this paper have listed in Table 1, which will be required to understand the problem formulation and other used descriptions. The equations have also described in detail throughout the respective sections based on their use.

3.3. Abbreviations Used

All the abbreviations used in this paper are summarized in Table 2 with their meaning.

3.4. Assumptions

All the assumptions that will be needed for this paper, are summarized in Table 3.

Table 1: **Details of Used Symbols**

Symbol	Meaning	Extra Information
M	Memory Bandwidth	-
i	Number of interconnections	-
$S(x)$	processing speed of a processor	-
$t(x)$	Execution time of problem	-
X	Total Performance of associate processors in a cluster	-
x	Individual Performance of interconnected processor	$x_i \cup x_{i+1} \in X$
N	Number of Nodes	-
W	Total Data Transfer Direction	-
θ	Data Transfer Direction of a single Processor	$\theta_i \cup \theta_{i+1} \in W$
μ	Ratio between performance of fast and slow CPU	-
c	Numeric coefficient for scaling the system	-
$A^W_\mu(N)$	Performance advantage of all Nodes (achievable gain)	$1.33 \lesssim A^W \lesssim 1.6$
L_p	Master fraction of Load and Processor	-
α, β	Normalize values	-
S_0	All possible solutions or slices	$S_0 = [S(N_0)[\cup S(N_1) \cup [S(N_2), \ldots, \cup S(N_x)]]]$

4. PROBLEM FORMULATION

In system designing, we have two scalability criteria: (1) weak scaling, (2) strong scaling. In strong scaling, the problem of memory congestion is not critical since it has been supported by the cache layer (L3)[43]. But in a weak scaling system, the memory congestion becomes critical. Therefore, this work focuses on congestion problems occurred during exchanging the data among different nodes through memory controllers and interconnecting links. Initially, the workload has distributed among different MPI applications which support parallel execution. These MPI applications, may also be able to send and receive messages during execution. The distribution of exchanged data can be irregular, so it can not always communicate in the same direction with all other MPI processes. One thing is notable that the time and amount of data transferred may vary based on their scalability. To find the performance of scalability of the architecture, one can use scalability matrix. The scalability matrix depends upon Algorithmic limitation, Bottlenecks, Startup Overhead, and communication.

The scalability matrix has two important criteria i.e. serial and parallel fractions. For a simple model in the scalability matrix, if the problem size (amount of work) is 1 then $s + p = 1$, which is runtime. s is the non-parallelizable part and p is the perfectly parallelizable fraction. To make it more comprehensive, Figure 1 shows a scenario where two groups of CPUs are connected directly with interlinking method and uses a memory controller framework, mentioned earlier in section 1. Based on the architecture shown in Figure 1, parallel applications can be described [43] as:

$$T^p_f = s + \frac{p}{N}. \tag{1}$$

In eq(1) T is a task, N is the number of workers or nodes deployed in a cluster, p is the perfectly parallelizable fraction and f is the fixed problem. Based on the eq(1), if the problem is of variable-sized, then the amount of running time we require, has shown in eq(2) and eq(3), respectively.

$$T^s_v = s + pN^\alpha \tag{2}$$

$$T^s_v = s + pN^{\alpha-1} \tag{3}$$

Table 2: **Details of Used Abbreviations**

Abbreviation	Meaning	Extra Information
NUMA	Non-uniform memory access	-
RCLB	Reducing congestion with load balancing	Proposed method
MPI	Message Passing Interface	Programming Language
NAS	Numerical Aerodynamic Simulation	to evaluate supercomputers
CLB	congestion with load balancing	memory based test
DRAM	”Dynamic Random Access Memory	Type of RAM
QP	Quilt Packaging	Interconnection Technique
CG	Conjugate Gradient kernel	-
FT	Fourier Transform kernel	-
LU	Lower-Upper Gauss-Seidel kernel	-
MG	Multi-Grid kernel	-
LB	Load Balancing	-
CUDA	Compute Unified Device Architecture	parallel computing platform and API
API	Application Programming Interface	-
GPU	Graphics Processing Unit	-
SCOTCH	Library	Software package and libraries for sequential and parallel graph partitioning, static mapping and clustering, sequential mesh and hyper-graph partitioning, and sequential and parallel sparse matrix block ordering
SSOR	Symmetric Successive Over-Relaxation	-
PDEs	Partial Differential Equations	-
LLC	Last-Level Cache	-
IPC	Instructions Per Cycle	-
NPB	Numerical Aerodynamic Simulation parallel benchmarks	-
OpenMP	Open Multi-Processing	API to supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran

Table 3: Assumptions for parallel computing architectures

Features	Assumptions in parallel computing
Priori Process coordination	Yes
Homogeneity on Network, Host and processes	Yes
Network speed	Fast
Network Reliability	High
Host speed	Fast
Host Reliability	High

So, based on scalability law, serial performance for fixed problem size is defined by eq(4),

$$P_f^s = \frac{s+p}{T_f^s} = 1. \tag{4}$$

and the parallel performance is calculated by eq(5),

$$P_f^p = \frac{s+p}{T_f^p(N)} = \frac{1}{s+\frac{1-s}{N}} \tag{5}$$

The eq(5) will depend upon the speedup. We have considered a CPU having p identical processors. A workload of size n has been executed over p. Let the processing speed of a processor is $S(x)$ for a problem size x. So the processing speed is formulated in eq(6) as:

$$S(x) = \frac{Size\ of\ Problem}{Executation\ time\ of\ the\ problem} = \frac{x}{t(x)} \tag{6}$$

Based on eq(5) and eq(6), we can say that the speedup can be super-linear and obtained from the ratio between the size of the problem and the time needed for execution of that problem. So, in this paper, we will need to calculate the speedup of its newly made virtual Cluster based on the architecture shown in figure 1. After that, on the basis of obtained speedup results, the next job is to calculate the memory congestion of the system for the proposed algorithms. From figure 1, If we consider the x as several processors, M as CPUs with memory controllers (in Bandwidth(%)), i as interconnections between CPUs, then the total performance is almost equal to the performance of interconnected processors, i.e. total memory congestion (traffic on memory controllers), represented as:

$$x \approx Mi : \forall x \in X \tag{7}$$

where X is the performance of associated processors in a cluster, which contain the number of nodes for all interconnected node with data transfer directions (θ). So, based on eq(7), the mapping of x depends upon individual memory controller bandwidth as $((x \in M_1) \times (x \in M_2) \times \ldots \times (x \in M_i))$ with data transfer direction and rate. Therefore, one can say that $(x \in X)$ is a vector which is almost equal to the multiplication of total memory and number of interconnected links. Since, all input instances are directed as in metric form based on the architecture shown in Figure 1. The column order of processors (mapped as: $(P_0, P_2, P_8, P_{10}), \ldots (P_5, P_7, P_{13}, P_{15})$) contains a single order of row wise distribution. Because of single order distribution, all the systems having data transfer direction (θ) be a strictly weak scaling system because, $(work \propto N)$ is a well-known derivation of Gustafson scaling, and in this case, $\forall(\theta \in W)$. For such systems, the speedup can be defined as:

$$S_x(N) = \frac{[s+(1-s)N]/(\mu+c(N))}{\mu^{-1}} = \frac{s+(1-s)N}{1+c(N)/\mu}. \tag{8}$$

From eq(8), in order to estimate the achievable gain, we have assumed that s is a negligible serial part which is lesser than 1, i.e., $c(N) > c(\mu N)/\mu$. However, the communication overhead may even increase with N but increment must be steady than linear. So, eq(8) leads to the eq(9), to find the achievable gain.

$$A_\mu^w(N) = \frac{[s+(1-s)\mu N][1+c(N)]}{[s+(1-s)N][\mu+c(\mu N)]} \tag{9}$$

The eq(9), introduces the concept of a positively robust scaling system to the proposed method, which needs to be optimize. From eq(9), if the increase is steady then coefficient c may be negative and the result may infer with some contingent value of (μN). So, this paper uses a non-negative square optimization technique. In this technique, the coefficients (c) are not allowed to become negative and smallest memory traffic will became traceable.

$$X = \min_{x_N} \left(\frac{1}{2}\|s - Mi\|^2\right) \text{ where, } x_N \geq 0, \forall N = 1,2,3,\ldots,n. \tag{10}$$

From eq(10), our objective is to reduce the memory congestion. As earlier mentioned that for any NUMA system, the data can be exchanged in two different ways: either on the same node or on the different nodes. In first case, the data exchange includes the communication traffic on memory controllers whereas, in the second case, data exchange includes the communication traffic on interconnected links and memory controllers both. By the definition of memory congestion if the communication traffic exceeds the bandwidth of memory controller, then memory congestion will happen. In our case, the communication pattern can vary based on the bandwidth of the memory controller because some MPI processes can induce substantial traffic on memory controllers as the bandwidth decreases from 100 to 0. As a result, the data which is to be exchanged become larger and a load-imbalance occurs between interconnecting link and memory controllers. Therefore we can easily say that the memory congestion on the same node is higher than a different node.

In order to reduce such memory congestions on the same node (on weak scaling systems), we need to allocate all the CPUs, even if at least one constraint is satisfied. So for the minimum X, this paper maps a better method to increase the cluster utilization and reduce the memory congestion.

5. PROPOSED ALGORITHM

5.1. Algorithms

Algorithm 1: Gathering the node information and communication pattern (NIC)

```
    Input: $x = Trans\_P\_times\_M(t,k,S_0,x)$
    Output: To predict dem_Value  and $Trans\_P\_times\_M(t,k,S_0,x)$
1   Procedure:
2   $[n,N,k,m,p]$ =non-zero coefficient associate with nodes $N_k$.
3   $Cluster\_Size \leftarrow \frac{N_k}{p}$ where: $1 \leq k \leq N$
4   $P = i\_times\_M(t,k,S_0,x)$
5   MPI_Bcast ($P_x$)
6   for $n = 1$ to $Cluster\_Size$ do
7       $N(k) = M^T\{P_{x(:,p(k))} + S_0(:,\alpha\frac{N_n}{p}) + \gamma(N_nP_k)\}$
8       compute partial $x$ with $hybrid\_model$
9   MPI_Gather($P_k$)
10  for $n = 1$ to $N_k$ do
11      $Load\_C(x) = \Sigma_{i=1}^{k} M_{p_i}$
12  Return $x$
```

The proposed algorithms 1 and 2, are based on MPI_Routines (MPI_Bcast, MPI_Gather and MPI_Reduce), which considers congestion and locality of application. The performance of the placement strategy has been performed in four steps which are as follows:

1. In the first step, the information about node topology, memory controllers, interconnected links has been retrieve by investigating the used hardware components. To perform this operation, we have used a Portable Hardware Locality (hwloc) tool, which is based on OpenMP. The Portable Hardware Locality tool assessed the existing system's topology and modeled them in a tree structure to retrieve the information about memory controllers location and links. These pieces of information are very useful for such mapping algorithms, who performs a match between MPI processes and system's cores. So, we have also used this information for our proposed Algorithm 1 and 2.

2. In the second step, the maximum consideration has been given to the MPI based applications. In this step, we have gathered the communication pattern by using the proposed algorithm 1. In this algorithm, a preliminarily profiling technique has also used to gather information about the target node. The profiling techniques have worked on two different domains: spatial and temporal domains. Our proposed method (Algorithm 1) considers both of the domains to gather communication patterns. In both of the cases, the execution of MPI processes and its ranking may vary. But in our case, it is considered to be constant for both of the domains. The communication pattern includes different parametric data: time-series, time-stamp tracing, number of messages and the size of all MPI applications. We are using on-line Dynamic Monitoring of MPI Communications tool developed by

Algorithm 2: Reducing congestion with load balancing (RCLB)

```
Input: $x = i\_times\_M(t,k,S_0,x)$
       $N : Total\ number\ of\ nodes; N_p : Total\ number\ of\ processes; X : Group\ of\ processes$
Output: To predict $dem\_Signal$ and $i\_times\_M(t,k,S_0,x)$
1  Procedure:
2  $[n,N,k,m,p]$ =non-zero coefficient associate with nodes N; $\forall(N) : 1 \leq N \leq n$
3  $Cluster\_Size \leftarrow \frac{N_k}{p} \forall(k) : 1 \leq k \leq N$
4  MPI_Bcast(m)
5  for $t \in 1,2,3,\dots$ to $Cluster\_Size$ do
6      $x(:,m(k)) \leftarrow P_{x(:,p(k))} + S_0(:,\alpha\frac{N_n}{p})$
7      compute partial $x$ with $hybrid\_model$
8      $L_p \leftarrow \{\alpha\frac{M_p}{\Sigma_{i=1}^{k} M_i} + \beta\frac{S_p}{\Sigma_{i=1}^{k} S_i}\}$
9      $L_{cluster} \leftarrow \Sigma_{i=1}^{k} M_{P_i}$
10 MPI_Reduce($\alpha$)
11 //To find best group of memory controller and interconnected link
12 $B \leftarrow create\ buckets(n,np)$
13 $G \leftarrow L_c = \Sigma_{i=1}^{N_{pc}} W_{p_i}$
14 $i \leftarrow 0$
15 while $size(B) < np$ and $i < size(G)$ do
16     $P \leftarrow sortedG[i]$
17     $P \leftarrow W_p = \alpha\frac{m_p}{\Sigma_{i=1}^{n} m_i} + \beta\frac{S_p}{\Sigma_{i=1}^{n} S_i}$
18     $distribute\ pairs(B, sorted\ P)$
19     $i \leftarrow i+1$
20 Store the aggregate result in $X$
21 Return $X$
```

George Bosilca et al [44], which is implemented for MPI and OpenMP. Although this tool is categorized into a low-level monitoring tool it is still in use. George Bosilca et al have improved again this monitoring framework by adding some more management layer. This tool is capable to trace the communication pattern in both of the cases: point-to-point or collectively. Moreover, it has some limitations, so that it can trace a limited number of processes together and the execution pattern will not be disturbed.

3. In the third step, to analyze the memory congestion, the time series data is required. Once the data have been obtained from the previous step, the investigation report identifies the place and time of the communication event. Generally, a single process is communicated by two applications. These applications are executed on the same node at the same time. Therefore the memory congestion occurs at the memory controller and its interconnected link. These pair of the memory controller and interconnection link can be identified by using the MPI process ranking technique and time trace of the communication event. In the proposed Algorithm 2, processes are grouped in the form of pairs, i.e. the total number of processes and the total number of nodes have mapped together. These groups can process upon different nodes at the same time and also communicate at different timestamps.

4. In the fourth step, we have applied the K-means clustering technique to obtain the group, because, at the beginning of the program, the group pair of processes are unknown. In this clustering method, the communication pattern needs an extra parameter, K, to specify the pairs. This parameter positively affects the traffic at memory controller point. Therefore it becomes essential to estimate the number of pairs of processes as a group in advance. Here the K-means clustering technique starts from K=2 and then starts increases towards estimation algorithm: goodness

of fit test. As the value of K increases, the time-stamp of pair of processes becomes smaller. At last, we have mapped the communication pattern between MPI ranks and cores in the same classified group based on the same time-stamp.

Based on these four steps the Algorithms (1 and 2) have implemented.

6. APPLYING THE ALGORITHM

6.1. *Algorithm 1:*

- *Gathering the node information and communication pattern (NIC)*

In **Algorithm 1**The objective function is to reduce the load on processors. The input function $x = i\_times\_M(t,k,S_0,x)$ has been passed as input to this algorithm where, M is the memory Bandwidth having time, iterative index, possible solution and individual performance of interconnected processors are the respective parameters. The procedure is to receive the **dem_Signal** with the input parameter with all associated nodes. Here the communication overhead may even increase with N but increment must be steady than linear. Thus, we have added the non-zero coefficients with associated nodes N. The next step is to decide the **Cluster_Size** which is a ratio of the number of working nodes and the number of processors they have. Then, in the next step, we have Broadcast the massages among all associated nodes with the help of **MPI_Bcast** routine. In the next step, we have calculated iteratively, the execution time of all processors and stored partially in the inter-communication node. We are calculating the load at all processor level based on their participation. Then, in the last loop, we have calculated the final load on the cluster. Then, the **MPI_Reduce** routine will call the load function through load coefficient to reduce the distributed load and store the aggregate result in a processor. While running this algorithm the communication pattern may vary several times based on kernel's configuration. As mentioned earlier that the placement strategy is based on MPI processes which are static in nature, so it supports ideal mapping for the execution. Therefore the memory controllers and interconnected links requires a priority list for optimization based on higher to lower traffics. In this paper, we have calculated two metrics. The first metric is based on a load of groups of pairs of processes. This metric is defined as a weighted metric W_p containing the sum of the normalized number of messages and their sizes based on every group. Where W_p is defined by below equation (11).

$$W_p = \alpha \frac{m_p}{\sum_{i=1}^{n} m_i} + \beta \frac{S_p}{\sum_{i=1}^{n} S_i} \tag{11}$$

where m_p is the number of messages exchanged by a pair p, s_p is the size of messages exchanged by a pair p, and n is the number of pairs in all groups. The normalized values of the number of messages and the size of messages are weighted by α and β, respectively.

The second metric is L_c , which is the load of a group c:

$$L_c = \sum_{i=1}^{N_{pc}} W_{p_i} \tag{12}$$

where N_{pc} is the number of pairs in a group, c, and W_{p_i} is the value of W_p for the i^{th} pair in a group, c.

6.2. *Algorithm 2:*

- *Reducing congestion with load balancing (RCLB)*

In **Algorithm 2**, we have applied **MPI_Bcast** on the total number of participated processors and calculate the distributed load on individual processors. at last, we have gathered the information about the processors and congestion at memory controllers and interconnected links. In this algorithm, we have used C libraries to integrate MPI_Bcast and **MPI_Gather** functions. After obtaining the number of pairs of processors from Algorithm 1, the RCLB algorithm performs a matching between the processes with and their respective cores. The matching factor ensures the reducing ratio and load balancing over memory controllers and interconnected links. The load balancing is done by using the above eq(11) and eq(12).

To measure MPI's performance, the main issue is the basis of its performance and the format of data distribution/processing. In our work, we have presented some important information about massive communication with huge data-set. We have presented the results based on parallel algorithms, widely used for the parallel implementation of various MPI-Routines. We have used optimized source code of algorithm, especially for the size. It is also capable of evaluating the data loading time and waiting time. The eq(13) defines the load of a cluster having X number of processors.

$$Load\_C(X) = \Sigma_{i=1}^{k} M_{p_i} \tag{13}$$

where k is the number of pairs in a group having a different combination of system components with x number of processors. M_{p_i} is the value of paired processors in that group.

The RCLB algorithm distributes a load of memory controllers by constructing buckets of size n. These buckets are represented by a node and the number of cores is defined as the capacity of the buckets. Based on the load, all processors are classified and placed inside each bucket. While placing the processes a greedy approach is used. Based on this technique no buckets are allowed to keep more than one process together which has same rank and a single process cannot use more than one bucket. In this procedure, the pairs of processors are executed in a priority order, i.e. from smallest to largest. The same procedure is also applied in the case of groups. This task is achieved by the steps (lines 15 to 18)[6, 7] written in Algorithm 2. Without violating the bucket's rules the optimization steps are mapped. Two processes from each group are selected and placed into a single bucket. This procedure is served in round-robin fashion.

6.3. *Complexity Analysis*

This section studies the run-time complexity of proposed methods. We evaluated the proposed model on the basis of two parameters:

- *Based on computational weight:* To always change the optimal solution, we need a continuous recursive process that runs with the help of Non-Negative Least Square Optimization Algorithm [45]. As per assumption that the data is running and loading continuously, a similar class of optimization algorithms takes almost constant running time for iteration. This step varies from basic to advance with input data at every level. Therefore, we can say that this model takes $O(max(N_n, N\theta_n))$ time.

- *Based on computing distribution of signals:* Let us assume that all the processors connected to a cluster are ideologically same. The complexity of proposed model based on achievable gain $(A_{\mu}^{w}(N))$ is O($\frac{N_n}{p}$). In each MPI process, different nodes are focused on the same ideology. The last load with x on MPI nodes takes $(N_n \times N)$ Rank, which is calculated by $x = M_y^T$. So in this case, the complexity will be the same as $Y = M_w$. As it is known that the complexity of equations in MPI programming can be calculated in HPC communication environment with a non-zero cost potential scenario which takes almost (O$\frac{N_n}{p} + N_n K$) time.

In general, for the limit of large bandwidth and small latency, the modified speedup is the favorable mode of operation for parallel computers with slow processors. So, we have found theoretical evidence that it can really be useful to build large machines with slow processors and our system is much more advanced than a traditional machine.

7. SIMULATION AND RESULT ANALYSIS

This section discusses the experimental results and validates the performance achieved. All the results presented in this paper are based on proposed method followed by two experiments.

- Based on Real system : The first experiment conducted on an Intel-based Xeon Phi 200 series based NUMA system.

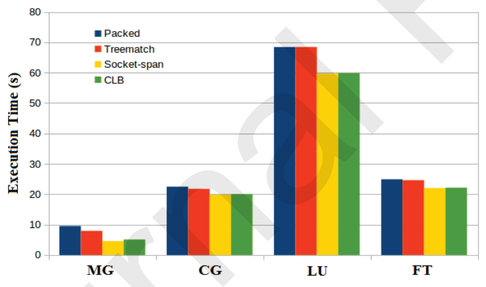
7.1. Experimental Setup

***Experimental Setup for first experiment*:** The Xeon Phi 7210 system is equipped with 64 cores, 128 GB of main memory and two Intel-based 7210 processors of 16 cores each running with a Max Turbo Frequency of 1.50 GHz. The system consists of four nodes, and each has eight cores with a processor frequency of 1.30 GHz. All nodes are interconnected via QP-Interconnect [46]. The connection topology of nodes is shown in figure 1. MPI and OpenMP are used to perform all the experiments. We have used four NAS parallel benchmarks(NPB) kernels in our experiments. They are listed below:

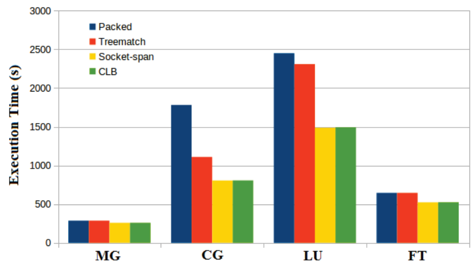
- Multi-Grid (MG) kernel [14]: Multi-Grid (MG) kernel of irregular communication.
- Conjugate Gradient (CG) kernel [14]: Conjugate Gradient (CG) kernel of irregular memory access and communication.
- Fourier Transform (FT) kernel [14]: Fourier Transform (FT) kernel of the all-to-all communication pattern.
- Lower-Upper (LU) Gauss-Seidel kernel [14]: Lower-Upper Gauss-Seidel kernel (LU) featuring a solver with irregular memory access.

We compared RCLB to three process placement algorithms: Packed; Socket-span and Treematch. Packed is very close to a logical identity and supports a greedy approach. For example: To map a constant value, one considers the calculation step of n numbers. Socket-span is used to map processes to the cores of different nodes. Procedures in the Socket-span policy. This type of facility has been provided by MPI in round-robin fashion. Treematch is also a placement strategy which maps processes based on their distribution between cores. We have used Treematch v0.4 for the task. Here the number of messages has the same degree of contribution to work w for proposed RCLB algorithm.

We have used to classes of NPB [14] i.e., C and D. In the NPB kernels these classes are used as average and large problem sizes. The systems used for this experiments has 28 processors, so the number of processes executing the MG, CG, and FT kernels [9, 14] be a power of 2. Therefore 2^4 processes have been used to executing the respective kernels. Only LU kernel will be executing more than 2^4 processes.



(a)



(b)

Figure 2: Average execution times of the NPB kernels for class C (a) and class D (b).

7.2. Performance Comparison of NPB kernels

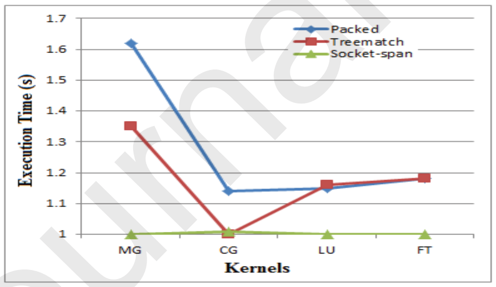
Figure 2(a) and (b) shows the average execution times of Class C and D of NPB kernels, which are average and Large in size. We have taken 50 sample executions and calculate the average execution time for kernels. The corresponding mean for the sample is calculated as 97.33% of confidence. The percentage of confidence level shows that the average execution time shown in figure 2 (a) and (b) are very close to the means of samples. In figure 2(a) RCLB shows a better result than stock-span which outperforms the packed and tree-match. Overall Stock-span performs better than all three Packed, Treematch and RCLB because it reduces the number of accessed communications. However, that node has still some remote-access communications which have a heavy calculation [47]. Such communications have assigned by Treematch to the same node, by increasing memory controllers of that node. Since memory controllers become unbalanced due to substantial connection, and as a result, the total execution time will increase. Therefore, as a result, RCLB performed better than packed and Treematch and prevents memory congestion by imposing a load imbalance policy. Here the best notable thing is that socket-span shows better in all kernels but not consider the communication patterns of the cores. Since it does not include the communication patterns, its mapping technique will distribute the data among all the nodes. That is why Socket-span reduces the load balance between two communicating nodes. Since

Table 4: **Mapping of processes of used placement methods for MG kernel in the order of (node:core).**

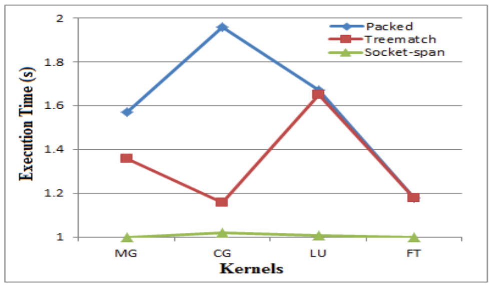
MPI Rank	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Packed	0:0	0:1	0:2	0:3	0:4	0:5	0:6	0:7	1:0	1:1	1:2	1:3	1:4	1:5	1:6	1:7
Treematch	2:0	2:1	2:2	2:3	2:4	2:5	2:6	2:7	3:0	3:1	3:2	3:3	3:4	3:5	3:6	3:7
Socket-span	0:1	0:1	1:1	1:1	0:2	0:2	1:3	1:3	0:4	0:4	1:5	1:5	0:6	0:6	1:7	1:7
RCLB	0:6	1:6	0:5	1:5	1:0	0:0	1:2	0:2	0:4	1:4	0:3	0:3	1:1	0:1	1:7	0:7

RCLB and socket-span have almost equal average execution time, so they are comparable and their mapping results are similar on all nodes. Table 4 shows the corresponding result of all four placement algorithms for MG kernel. In this table, all the MPI processes are represented by their rank number and cores are represented by their id number starting from 0 to 15. Its seems that the default core mapping is different from Socket-span and RCLB but actually their mapping technique is almost the same. In Socket-span, MPI processes search for their neighboring process from the different node and allocate in a binary: (0,1,0,1). The RCLB method also allocates the core and node in binary format but the sequencing is different:(0,1,0,1,1,0,1,0) . This sequencing is likely to be allocated to different placement methods. Finally, the result shows that the RCLB and Socket-span both are almost same.

When the problem size is large then NPB uses class D. The results of Class D is shown in figure 2(b). Based on the results the Socket-span and RCLB show better performance for all four kernels. The minimum performance is obtained for MG kernel and maximum performance for LU kernel. The minimum performance indicates that the amount of data exchanges between interconnected nodes are very less. Similarly, The maximum performance indicates that the amount of data exchanges between interconnected nodes are very high. The RCLB and Socket-span both, shows almost the same performance with respect to all benchmarks: MG; CG; LU; FT [9, 14]. It means the performance gain is remain unchanged even if the problem is big. In this case, the communication pattern follows the all-to-all communication path for all nodes. Therefore, the remote access communication will not be large and as a result, the process does not affect the performance.



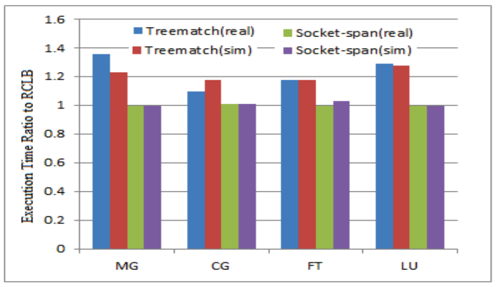
(a)



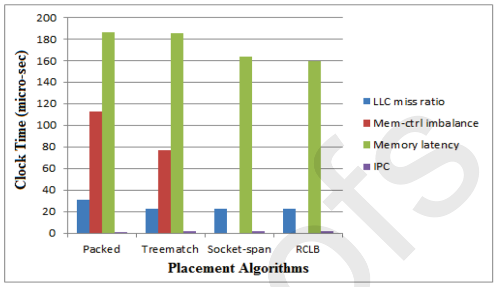
(b)

Figure 3: Normalized execution times of the NPB kernels for class C (a) and class D (b).

Figure 3(a) and (b) shows the normalized execution time for class C and class D problem sizes. In this figure, the RCLB acts as a baseline and the rest of the methods are derived. Therefore, the results are comparable to RCLB. From the respective figure 3, we can strongly say that each method needs a longer time than the baseline method. The Socket-span method shows a Lower and equal time graph with respect to RCLB, which is a good sign for better. Here one thing is observable that, whenever the problem size has increased, the concern methods: Packed and Treematch, has increases their execution time. This happens because the load imbalance increases among all nodes as the problem size increases. So, Packed and Treematch cannot prevent such cases. Therefore, we can say that RCLB can definitely reduce the load imbalance problem among all interconnected nodes. The figure 3(a) and (b) shows a better performance even if all nodes presented in the system has a large memory congestion impact.



(a)



(b)

Figure 4: Result verification (a) and Memory congestion effect (b) for MG Kernels.

In figure 4(a), The MG kernel has been evaluated on Linux Perf-profiling tool for the memory controller activities. One more Intel-based profiling tool (Uncore) has been used to profile the used CPU's memory controller activities. Based on both of the profiling tools we have observed that the memory controller imbalancing graph is very low in the case of Socket-span and RCLB. The rest all placements have a high graph reporting, thus we can say that all the performance differences are due to memory congestion. The figure 4(b) illustrate the performance difference of each placement methods based on some supporting datasets. In this figure we have used four benchmarked metrics for analysis: LLC miss, Memory controller imbalance, Memory Latency and IPC. The LLC miss ratio is highest for Packed and lowest for Socket-span. Here one thing is observable the LLC miss ratio is almost equal between Socket-span and RCLB, which is good sign to work further. The higher value shows a higher risk of memory congestion for that placement method. The higher execution time shows that a higher increase in memory latency because the CPU gets a long wait to fulfill the LLC miss. The larger graph of IPC shows a higher performance for corresponding placement methods. Thus the figure 4(b) shows the validity of obtained result for MG kernel. The final conclusion of figure 4(b) is based on three observations.

1. The Packed has the highest LLC miss and longest memory latency because it has the lowest IPC.
2. The memory latency of RCLB and Socket-span is small and they have a higher IPC than other placement methods. As we have earlier mentioned that the memory latency increases with the load imbalance of the memory controllers.
3. The RCLB and Socket-span both have been reducing the memory congestion by preventing the memory controllers load imbalance.

7.3. Simulation

In order to calculate the congestion effects on different placement methods for class C and class D problems, a large scale simulation have conducted. From figure 4 (a)and (b) it is clear that the RCLB can achieve comparable performance with the method Socket-span. From the result 4(a) it is also clear that whenever the number of processors becomes larger then the socket-span method can not be able to maintain the memory controller imbalance. The reason is Socket-span method uses communication patterns of applications. One thing is notable that we have used SMPI simulator v3.20 for the experiments. This simulator is capable to capture the effect of network congestion with communication distance. In this simulation environment the kernels are executed on the virtual platform but behave like a target platform.

7.3.1. Simulation setup

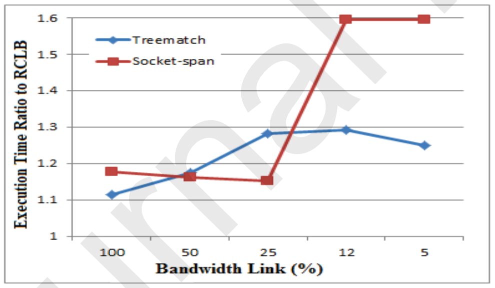
SimGrid simulator is handy for Network clusters. To simulate the memory congestion effects on NUMA systems, it provides a virtual platform with imitating topology of modern systems. The resources configured on SimGrid correctly represents the cores and memory controllers. The cores interconnected with memory controller links having a higher bandwidth and lower latency. The communication between two nodes uses a memory controller link, and the interaction between two cores uses default interconnection link.

7.3.2. Simulation Verification

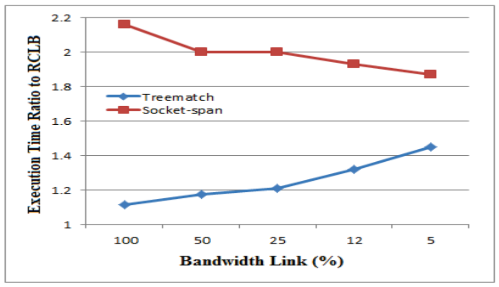
The accuracy of our simulation result verified by comparing with real-time NUMA systems. The figure 4(a) shows the simulation results, where the performance of the system is defined by a ratio of the execution time of that system and other systems with the same method. From the figure, we have observed that almost in every case except FT, the Socket-span for real and simulated both are almost same. As we have discussed earlier the communication pattern is different for Socket-span method. In other hands for FT method, the Treematch (real and simulated both) is almost same, but for other methods it has a good variation, which shows that the communication pattern depends on the memory imbalance. The platform has 8 memory controller link and 64 hosts of Xeon Phi 200 series processors. Each memory controller connects the hosts having the same resources. With the help of Intel memory latency checker[1], we have configured the latency and bandwidth of each controller link. As mentioned earlier, the congestion is very responsive to the bandwidth of the memory controllers, so in this paper, we have reduced the bandwidth for making an oppressive congestion environment. We kept the bandwidth ratio as it is, so that while execution of any application, more massive congestion will happen. At last, we have configured the environment according to OpenMP and MPI for mutual communication between nodes. The respective verification results are shown in figure 4(a). The individual verification results have shown in figure 4(a). From the figure 4, the execution time is similar to the systems (real and simulated). The difference between the execution time of actual and simulated experiments are notable. Therefore we can say that the simulation is accurate enough to the real system environment to demonstrate the memory congestion effect of the proposed method for significant problems.

7.4. Performance comparison of simulated results

After the verification of the accuracy of simulated results, the performance evaluation, and analysis have been conducted on different bandwidth of memory controllers. From the other results obtained in previous figures 2, 3,and 4, it is clear that Packed is the slowest method based on execution time. Thus we have skipped the Packed placement method in this result. The topology for the simulation has been shown in figure 1. The topology of a four-node NUMA system, shown in figure 1, has been used for the simulation. In this topology, the QP-Interconnect has four sockets with a memory controller and eight interconnected links. In this way, the computing node has been configured. Therefore to make use of all processors, we have executed 64 MPI processes. In the real and simulation method, the baseline bandwidth was the same. Then we have decreased the bandwidth in a order of 50%, 25%, 12%, and 5%.



(a)



(b)

Figure 5: Simulation Result in terms of execution time with respect to RCLB, CG (a) and FT (b) Kernels.

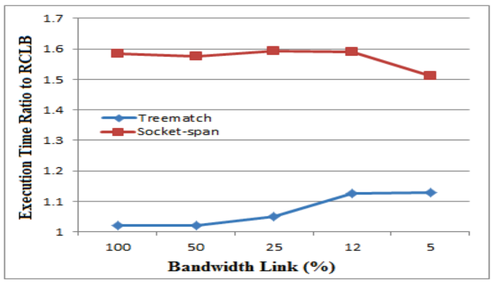
Figure 5(a) and (b) shows the evaluated results of CG and FT kernels for different memory congestion case. These results have been evaluated on SimGrid tracing and analyzing features. These features also provide the facility to find and analyses the latencies and utilization.

Figure 5(a) demonstrates the performance of the proposed method concerning CG kernel. In this figure, the proposed method shows a better result based on an increase in bandwidth(%). As mentioned earlier that the Communication pattern of CG kernel is irregular, so, Socket-span method shows better result compared to other methods. Since Socket-span suffers from high latencies of slowly communicating processes, so, without considering the communication

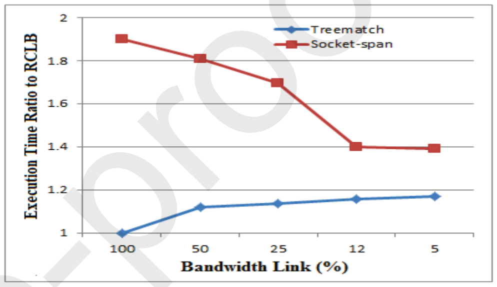
[1]https://software.intel.com/en-us/articles/intelr-memory-latency-checker

pattern, the Socket-span method distributes all the jobs among all nodes. In this figure, the Treematch also distribute the slowly communicating processes among all nodes. Thus the memory congestion increases the bandwidth and memory controller become smaller. Therefore the proposed method reduces the congestion by preventing the imbalanced memory load. Finally, our proposed method achieves the best performance, among other techniques.

Figure 5(b) presents the performance results of the proposed method on FT kernels. From this result, it is clear that the Socket-span need two times longer execution time than the proposed method. The FT kernel operates all to all communication pattern. This is the reason why Socket-span sufferers from higher latencies. In the case of Treematch, this method reduces communication locality by reducing remote-accesses. Similarly, the proposed method also reduces the number of remote-accesses by considering communication locality. Only the difference between Treematch and proposed method execution time is, as the bandwidth decreases, the performance increases. Thus by considering these results, we can say that the proposed method can reduce the memory congestion.



(a)



(b)

Figure 6: Simulation Result in terms of execution time with respect to RCLB, LU (a) and MG (b) Kernels.

Similar patterns of execution time ratio with the proposed method has been shown in figure 6(a) and (b). The figure 6(a) shows the performance of the proposed method on LU kernel and (b) shows the performance results of the proposed method on MG kernel. In earlier sections, we have mentioned that the MG kernel also works, on irregular communication pattern and LU kernel also do the same. The traffic on MG kernel is larger than LU kernel. So the Socket-span again sufferers from higher latency and execution-time increases between all communicating nodes. As a result, the congestion level becomes too low and hence the load imbalance among all memory controllers also reduced. Here in figure 6(a) and (b), the execution time of Treematch and the proposed method is almost the same and they are achieving better execution time than Socket-span.

7.5. Performance based on Locality and Congestion

For the NUMA systems performance, the benchmarks can provide an important picture. NAS benchmarks are universally validated and accepted. from this benchmark, the Princeton application repository is the best example of shared memory architecture and best suitable for NUMA effects because it uses more than 30% of CPU. We have conducted our experiment on AMD server having four quad-core CPUs. The detailed experiment and configuration have been explained in the later section.

The first experiment was done to find the effect of memory congestion on simple systems. To conduct this experiment, the benchmarks were run to make a limited pressure on memory controllers and interconnected links through a single thread. Based on findings a comparison was done between remote configurations and local configurations. The local configurations have their own memory and generated thread, runs on the same node whereas, at remote configurations, the memory has been shared by a different node through a standard linux tool.

The difference between Packed (P) and Socket-span (S) is shown in Table 5. In this table, all the matrices are based on multi-threaded theory. Based on two greedy approaches: Packed and Socket-span, we have shown absolute performance for different matrices. The table shows a comparison between both policies based on the performance difference between the best and worst policy for the used benchmark. The observation makes a clear conclusion that the percentage of Memory-controller imbalance is best for all applications.

Table 5: **Traffic congestion effects**

	Packed		Socket-span	
	Best	Worst	Best	Worst
Local-access (%)	23	23	23	27
Memory latency	336	1084	471	760
Interconnect imbalance (%)	18	67	12	73
Interconnect usage (%)	47	35	68	33
Memory-controller imbalance (%)	**6**	117	**3**	125
Inter Process Communication	0.23	0.11	0.49	0.29

8. CONCLUSION

In this paper, we have proposed a method (RCLB), which is capable to reduce the memory congestion with reconcile locality. It prevents the load imbalancing on memory controllers. It identifies the data traffic applying through clusters using MPI communication patterns. The evaluation results show the ability of RCLB in comparison with two greedy approaches: packed and stack-span. The proposed method has been tested with four kernels of NAS Parallel Benchmarks and found better than the other methods. We have also classified the characteristics of NUMA architectures at the hardware level by proposing a new architecture model. The proposed algorithm has been applied to our new architecture and found that the proposed method is able to fully utilize the maximum capacity of multi-cores. Such evaluation studies have not done quantitatively earlier for this purpose (reducing memory congestion). Finally, we have concluded that the improvement in recent hardware does not eliminate the NUMA bottleneck. The real-world testbed simulations and experiments show that the proposed method has achieved a bandwidth improvement of about 12.65% to 23% in memory-based tests.

This work can be extended in several directions. We plan to test the proposed algorithm on a large cluster. then we will find and analyze the contention behavior tasks of placement methods. This is to make the thread mapping module not only NUMA-aware but also contention-aware. Furthermore, thread migration algorithms will be considered in a comprehensive resource mapping module. Furthermore, we plan to find and analyze the energy consumption, load balancing of the proposed method. This paper can be implemented on the basis of thread scheduling in a hybrid programming environment with MPI and OpenMP.

References

[1] T. A. Engel, A. S. Charao, M. kirsch Pinbeiro, L.-A. Steffenel, Performance improvement of data mining in weka through multicore gpu acceleration, Ambient Intell Human comput 6 (5) (2015) 377–390.

[2] J. F. M. D. A. F. V. Q. R. L. Fabien Gaud, Baptiste Lepers, M. Roth, Optimizing numa systems applications with carrefour, Communications of the ACM 58 (12) (2015) 59–66.

[3] D. H. R. S. Daniel Molka, Main memory and cache performance of intel sandy bridge and amd bulldozer, ACM Workshop on Memory Systems Performance and Correctness (MSPC14), 17 (9) (2015) 1–10.

[4] S. e. a. Boyd-Wickizer, Corey: An operating system for many cores, 8th Usenix Symposium on Operating Systems and Design (2008) 43–57.

[5] S. M. N. F. B. G. G. M. S. T. Francois Broquedis, Jerome Clet-Ortega, R. Namyst, hwloc: a generic framework for managing hardware affinities in hpc applications, 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (2010) 180–186.

[6] K. K. R. E. Mulya Agung, Muhammad Alfian Amrizal, H. Takizawa, A memory congestion-aware mpi process placement for modern numa systems, IEEE 24th International Conference on High Performance Computing (HiPC) (2017) 152–161.

[7] R. E. Mulya Agung, Muhammad Alfian Amrizal, H. Takizawa, An automatic mpi process mapping method considering locality and memory congestion on numa systems, IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC) (2019) 17–24.

[8] R. A. M. R. J. S. Dimitrios Ziakas, Allen Baum, Intel quickpath interconnect architectural features supporting scalable system architectures, 18th IEEE Symposium on High Performance Interconnects (2010) 1–6.

[9] D. Bailey, Nas parallel benchmarks, RNR Technical Report.

[10] J. Corbet, Autonuma: The other approach to numa scheduling, LWN.net.

[11] G. R. David, T., V. Trigonakis, Everything you always wanted to know about synchronization but were afraid to ask, 24 th ACM Symposium on Operating Systems Principles (2013) 33–48.

[12] C. Lameter, An overview of non-uniform memory access, Commun. ACM 59 (9) (2013) 59–65.

[13] F. Cappello, D. Etiemble, Mpi versus mpi+openmp on the ibm sp for the nas benchmarks, Proceedings of the IEEE/ACM SC2000 Conference (SC00) (2000) 1–12.

[14] J. B. D. B. R. C. L. D. R. F. P. F. T. L. R. S. H. S. V. V. D.H. Bailey, E. Barszcz, S. Weeratunga, The nas parallel benchmarks, International Journal of High Performance Computing Applications (1991) 63–74.

[15] D. Y. Tan Li a, Yufei Ren, S. Jin, Analysis of numa effects in modern multicore systems for the design of high-performance data transfer applications, Future Generation Computer Systems 74 (5) (2017) 41–50.

[16] G. M. Emmanuel Jeannot, F. Tessier, Process placement in multicore clusters: Algorithmic issues and practical techniques, IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS 25 (4) (2014) 993–1002.

[17] L. V. K. A. Bhatele, S. Kumar, Dynamic topology aware load balancing algorithms for molecular dynamics applications, 23rd international conference on Supercomputing, ACM (2009) 110–116.

[18] D. C. C. M. A. B. P. O. A. N. F. . B. J.-F. . M. L. V. K. Aercio L. Pilla, Christiane Pousa Ribeiro, A hierarchical approach for load balancing onparallel multi-core systems, ICPP,12 (2012) 1–10.

[19] C. P. R. P. C. F. B. B. G. L. L. Pilla, P. O. A. Navaux, J. F. Mhaut, Asymptotically optimal load balancing for hierarchical multi-core systems, 18th International Conference on Parallel and Distributed Systems (2012) 236–243.

[20] G. M. F. T. E. Jeannot, E. Meneses, G. Zheng, Communication and topology-aware load balancing in charm++ with treematch, IEEE International Conference on Cluster Computing (CLUSTER) (2013) 1–8.

[21] G. Mercier, J. Clet-Ortega, Towards an efficient process placement policy for mpi applications in multicore environments, Advances in Parallel Virtual Machine and Message Passing Interface,LNCS, Springer 5759.

[22] Z. Majo, T. R. Gross, Memory management in numa multicore systems: trapped between cache contention and interconnect overhead, International Symposium on Memory Management, ACM (2011) 11–20.

[23] J. P. A. F. C. L. M. E. R. Rodrigues, P. O. A. Navaux, L. V. Kale, A comparative analysis of load balancing algorithms applied to a weather forecast model, Computer Architecture and High Performance Computing (2010) 71–78.

[24] F. Pellegrini, J. Roman, Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, International Conference on High-Performance Computing and Networking (HPCN 1996). Springer (1996) 493–498.

[25] E. Jeannot, G. Mercier, Near-optimal placement of mpi processes on hierarchical numa architectures, Euro-Par 2010 - Parallel Processing, LNCS, Springer 6272.

[26] J. H. B. R. H. Chen, W. Chen, H. Kuhn, Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters,, Proceedings of the 20th annual international conference on Supercomputing. ACM (2006) 353–360.

[27] W. C. J. Zhang, J. Zhai, W. Zheng, Process mapping for mpi collective communications, European Conference on Parallel Processing. Springer (2009) 81–92.

[28] D. P. Mahato, R. S. Singh, Balanced task allocation in the on-demand computing-based transaction processing system using social spider optimization, Concurrency Computat: Pract Exper. 29 (e4214) (2017) 1–26.

[29] S. H. M. Iman Faraji, A. Afsahi, Topology-aware gpu selection on multi-gpu nodes, IEEE International Parallel and Distributed Processing Symposium Workshops, USA (2016) 712–720.

[30] N. M. Upadhyay, R. S. Singh, Performance evaluation of classification algorithm in weka using parallel performance profiling and computing technique, Fifth IEEE International Conference on Parallel, Distributed and Grid Computing (PDGC) (2018) 522–527.

[31] S. Greengard, Gpus reshape computing, Communincation, ACM 59 (9) (2016) 14–16.

[32] D. O. E. B. Brendan Burns, Brian Grant, J. Wilkes, Borg, omega, and kubernetes, ACM Queue (2016) 70–93.

[33] R. G. P. R. David Lo, Liqun Cheng, C. Kozyrakis, Improving resource efficiency at scale with heracles, ACM Trans. Comput. Syst. 34 (2) (2016) 116–149.

[34] P. A. Akshat Verma, A. Neogi, Power-aware dynamic placement of hpc applications, 22nd Annual International Conference on Supercomputing, ACM (2008) 175–184.

[35] N. Y. Alexandru Iosup, D. Epema, On the performance variability of production cloud services, 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. IEEE Computer Society, USA (2011) 104–113.

[36] S. J. Yaakoub El-Khamra, Hyunjoo Kim, M. Parashar, Exploring the performance fluctuations of hpc workloads on clouds, IEEE Second International Conference on Cloud Computing Technology and Science. IEEE Computer Society, USA (2010) 383–387.

[37] A. J. R. A. M. T. K. G. H. L. O. M. Onur Kayiran, Nachiappan Chidambaram Nachiappan, C. R. Das, Managing gpu concurrency in heterogeneous architectures, 47th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, USA (2014) 114–126.

[38] J. Wu, B. Hong, Collocating cpu-only jobs with gpu-assisted jobs on gpu-assisted hpc, 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (2013) 19–24.

[39] J. F. F. G. R. L. B. L. V. Q. M. Dashti, A. Fedorova, M. Roth, Traffic management: a holistic approach to memory placement on numa systems, ACM SIGPLAN Notices 48 (4) (2013) 381–394.

[40] N. Beckmann, D. Sanchez, Modeling cache performance beyond lru, 22nd International Symposium on HighPerformance Computer Architecture (HPCA), (2016) 1–12.

[41] S. W. K. Vikas Agarwal, M.S.Hrishikesh, D. Burger, Clock rate versus ipc:the end of the road for conventional micro architectures, 27th annual international symposium on Computer architecture (2000) 1–13.

[42] C.-H. Cheng, Design example of useful memory latency for developing a hazard preventive pipeline high-performance embedded-microprocessor, Advanced VLSI Design Methodologies for Emerging Industrial Multimedia and Communication Applications, Hindawi (2013) 1–10.

[43] G. W. Georg Hager, Introduction to high performance computing for scientists and engineers, CRC Press (2010) 1–356.

[44] E. J. G. M. G. Bosilca, C. Foyer, G. Papaure, Online dynamic monitoring of mpi communications, European Conference on Parallel Processing, Springer (2017) 49–62.

[45] G. K. X.Tang, K.Li, F.Wu, A stochastic scheduling algorithm for precedence constrained tasks on grid, Future Generation Computer Systems 27 (8) (2011) 1083–1091.

[46] R. A. M. D. Ziakas, A. Baum, R. J. Safranek, Intel quickpath interconnect architectural features supporting scalable system architectures, 18th IEEE Symposium on High Performance Interconnects (2010) 1–6.

[47] G. M. M. Q. M. S. A. Degomme, A. Legrand, F. Suter, Simulating mpi applications: the smpi approach, IEEE Transactions on Parallel and Distributed Systems 28.