Chapter 4

Memory Congestion Reduction in Multi-core Systems

In contemporary multi-core systems, the proliferation of cores exacerbates significant memory congestion issues at shared caches and memory controllers. Thus, this chapter[1] explores strategies for alleviating congestion through the Load Balancing for Memory Congestion Reduction (RCLB) technique. RCLB aims to mitigate memory congestion by optimizing data locality and ensuring balanced loads across memory controllers. It achieves this by analyzing MPI communication patterns within clusters during runtime with minimal modifications to profiling techniques. Evaluation using NAS Parallel Benchmarks (NPB) kernels demonstrates the effectiveness of the proposed approach. Experimental results indicate bandwidth improvements ranging from 12.65% to 23% in memory-intensive tests.

4.1 Introduction

The system design faces scalability challenges, particularly in weak scaling scenarios, where memory congestion becomes critical during data exchange. The data exchange happens between two or more nodes simultaneously through memory controllers and

[1] The work is published by Navin Mani Upadhyay and Ravi Shankar Singh in the Journal of King Saud University - Computer and Information Science, titled "An Effective Scheme for Memory Congestion Reduction in Multi-Core Environments," Volume 34, Number 6, Pages 3864-3877, in 2022. [SCI]

interconnecting links. In multi-core clusters, the challenge of memory congestion poses a significant obstacle to achieving optimal performance and scalability. Memory congestion arises due to the simultaneous and potentially conflicting demands for data access from multiple cores sharing a common memory subsystem. This issue becomes particularly pronounced in parallel processing scenarios, where efficient data sharing and communication are essential. The irregular data distribution among multi-core systems requires additional parallelization, which leads to varying communication directions and data transfer rates.

Mitigating memory congestion in multi-core clusters is critical, requiring solutions to balance simultaneous data access demands among cores and optimize communication efficiency. The challenge involves addressing conflicts in shared memory subsystems during parallel processing, aiming for enhanced system performance and resource utilization.

In system design, scalability is evaluated through two main criteria: **weak scaling** and **strong scaling**. In strong scaling, memory congestion is less critical due to the support of the L3 cache layer [101]. However, in weak scaling systems, memory congestion becomes a significant issue. This chapter addresses the congestion problems that arise during data exchanges among different nodes via memory controllers and interconnecting links. Initially, workloads are distributed among various MPI applications to support parallel execution. These applications can send and receive messages during execution, resulting in irregular data distribution, which prevents uniform communication across all MPI processes. Consequently, the time and amount of data transferred vary based on the scalability of the system. To evaluate the performance, this research utilizes a scalability matrix, accounting for algorithmic limitations, bottlenecks, startup overhead, and communication.

The scalability matrix focuses on two critical components: serial and parallel fractions. For a simple model in this matrix, if the problem size is 1, then the serial and parallel parts sum to one ($s + p = 1$), where $\boldsymbol{s}$ is the non-parallelizable portion and $\boldsymbol{p}$ is the perfectly parallelizable fraction. To illustrate this, Figure 3.1 depicts a scenario where two groups of CPUs are interconnected using a memory controller framework, as discussed in Chapter 3. Based on the architecture in Figure 3.1, parallel applications can be analyzed and scaled using Amdahl's Law and Gustafson's Law [101].

This research investigates memory congestion and calculates the speedup of a newly designed virtual cluster based on the architecture shown in Figure 3.1. Using the obtained

speedup results, we then calculate the system's memory congestion for the proposed algorithms. Considering P as a set of processors, M as a system with memory controllers (in Bandwidth (%)), and i as the interconnections between cores, the total performance is a product of the performance of interconnected processors, a single core's performance, and system efficiency, equivalent to the total memory congestion and referred to as traffic on memory controllers. This relationship is represented as Eq. (4.1).

$$P \approx M_i : \forall i \in P \qquad \text{Eq. (4.1)}$$

Here, $\boldsymbol{P}$ is the total number of processors in a multi-core cluster. Given that all nodes are interconnected, data transfer direction $(\boldsymbol{\theta})$ must be considered. Thus, according to Eq. (4.1), the mapping of $\boldsymbol{P}$ depends on individual memory controller bandwidth, expressed as $((\boldsymbol{P} \in \boldsymbol{M_1}) \times (\boldsymbol{P} \in \boldsymbol{M_2}) \times \ldots \times (\boldsymbol{P} \in \boldsymbol{M_i}))$ with respect to data transfer direction and rate. Consequently, $(\boldsymbol{P} \in \boldsymbol{M})$ is a vector approximately equal to the product of total memory and the number of interconnected links.

All input parameters are represented in a matrix form based on the architecture shown in Figure 3.1. The column order of processors, mapped in terms of $((\boldsymbol{P_0}, \boldsymbol{P_2}, \boldsymbol{P_8}, \boldsymbol{P_{10}}), \ldots (\boldsymbol{P_5}, \boldsymbol{P_7}, \boldsymbol{P_{13}}, \boldsymbol{P_{15}}))$, contains a single-order row-wise distribution. Due to this distribution, systems with data transfer direction $(\boldsymbol{\theta})$ are strictly weak scaling systems, following Gustafson's scaling law where $(\boldsymbol{work} \propto \boldsymbol{N})$. For such systems, the speedup can be defined in Eq. (4.2):

$$S_x(N) = \frac{[s+(1-s)N]/(\mu + c(N))}{\mu^{-1}} = \frac{s+(1-s)N}{1+c(N)/\mu} \qquad \text{Eq. (4.2)}$$

From Eq. (4.2), to estimate the achievable gain, this research assumes $\boldsymbol{s}$ is a negligible serial part, less than $\mathbf{1}$, i.e., $\boldsymbol{c(N) > c(\mu N)/\mu}$. Though communication overhead may increase with N, it should do so more gradually than linearly. Therefore, Eq. (4.2) leads to Eq. (4.3) to find the achievable gain.

$$A_{\mu}^{w}(N) = \frac{[s+(1-s)\mu N][1+c(N)]}{[s+(1-s)N][\mu + c(\mu N)]} \qquad \text{Eq. (4.3)}$$

Eq. (4.3) introduces the concept of positively robust scaling systems to the proposed method, requiring optimization. If the increase is steady, the coefficient c may be negative, inferring results with some contingent value of (μN). Hence, this research employs a non-negative square optimization technique, where coefficients (c) cannot be negative, ensuring traceable memory traffic with Eq. (4.4) below.

$$X = \min_{xN} \left(\frac{1}{2} \|s - Mi\|^2 \right) \text{ where, } xN \geq 0, \forall N = 1, 2, 3, \ldots, n. \qquad \text{Eq. (4.4)}$$

The objective from Eq. (4.4) is to reduce memory congestion. As previously mentioned, data in any multi-core cluster can be exchanged in two ways: within the same node or across different nodes. The first scenario involves communication traffic on memory controllers, while the second includes traffic on interconnected links and memory controllers. Memory congestion occurs if communication traffic exceeds the memory controller's bandwidth. This research considers variations in communication traffic due to bandwidth changes, which can significantly affect data exchange and cause load imbalances between interconnecting links and memory controllers. Therefore, it is possible to predict memory congestion levels on the same node versus different nodes.

This research thus develops a method to improve cluster utilization and reduce memory congestion, utilizing cache partitioning techniques to manage and optimize data traffic effectively.

4.2 Proposed Algorithm

As discussed earlier, this research proposes two algorithms: Node Information Communication (NIC) and Reducing congestion with load balancing (RCLB).

4.2.1 Gathering the node information and communication pattern

The proposed Algorithm 2 leverages MPI routines (MPI_Bcast, MPI_Gather, and MPI_Reduce) to address congestion and application locality through an innovative cache

Algorithm 2 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 $Trans_P_times_M(t,k,S_0,x)$
Procedure:
$[n,N,k,m,p]$ =non-zero coefficient associate with nodes N_k.
$Cluster_Size \leftarrow \frac{N_k}{p}$ *where:* $1 \leq k \leq N$
$P = i_times_M(t,k,S_0,x)$
MPI_Bcast (P_x)
for $n = 1$ *to* $Cluster_Size$ **do**
  $N(k) = M^T\{P_{x(:,p(k))} + S_0(:,\alpha\frac{N_n}{p}) + \gamma(N_nP_k)\}$
  compute partial x with *hybrid_model*
end
MPI_Gather(P_k)
for $n = 1$ *to* N_k **do**
  $Load_C(x) = \Sigma_{i=1}^{k} M_{p_i}$
end
Return x

partitioning technique. The strategy for collecting node information is executed in four detailed steps:

1. First, the topology of nodes, memory controllers, and interconnected links is examined by assessing the hardware components utilized. This is accomplished using the Portable Hardware Locality tool (hwloc), which evaluates the system's topology and organizes it into a tree structure. This structure reveals the locations of memory controllers and their interconnections, providing crucial data for mapping algorithms that align MPI processes with system cores effectively.

2. The second step prioritizes MPI-based applications by using the proposed Algorithm 2 to gather communication patterns. This algorithm employs a profiling technique to collect information from two key domains: spatial and temporal. Our method considers both domains to compile comprehensive communication patterns, which include parametric data such as time series, timestamp tracing, the number of messages, and the size of all MPI applications. For this, we use the online Dynamic Monitoring of MPI Communications tool developed by George Bosilca et al. [95], which, despite being categorized as a low-level monitoring tool, remains relevant.

This tool can trace communication patterns in both point-to-point and collective cases, though it has some limitations in tracing a limited number of processes simultaneously without disturbing the execution pattern.

3. In the third step, the gathered communication patterns are analyzed to identify hotspots and congestion points within the shared cache and memory controllers. By understanding these patterns, we can partition the cache more effectively, ensuring that high-traffic data paths are allocated sufficient cache resources to reduce contention and improve performance.

4. Finally, the fourth step involves dynamically adjusting the cache partitioning based on real-time monitoring of memory access patterns. This adaptive approach ensures that as the workload evolves, the system can reallocate cache resources to maintain optimal performance, further minimizing memory congestion and enhancing the overall efficiency of multi-core systems.

Overall, the integration of cache partitioning with the proposed Algorithm 2 and MPI routines provides a robust solution for addressing memory congestion and optimizing application locality in multi-core environments.

4.2.2 Reducing congestion with load balancing (RCLB) Algorithm

Once the data has been obtained from the previous step, the investigation report identifies the location and timing of communication events, referred to as time-series data. This dataset is essential for the third step. In a multi-core cluster scenario, a single process typically communicates with multiple applications, leading to memory congestion at the memory controller and causing a bottleneck at its interconnected links. Identifying these congestion points is crucial. By utilizing the MPI process ranking and time tracing technique, it is possible to pinpoint these congestion points to a certain extent. In the proposed Algorithm 3, processes are grouped into pairs (i.e., the total number of processes and nodes) and mapped together. These groups can be distributed across different nodes simultaneously and communicate at distinct timestamps.

To enhance cache efficiency and manage memory congestion, we employ a cache partitioning technique. This method segments the cache space to ensure that memory

Algorithm 3 Reducing congestion with load balancing (RCLB)

Input: $x = i_times_M(t,k,S_0,x)$

$N : Total\ number\ of\ nodes; N_p : Total\ number\ of\ processes; X : Group\ of\ proc.$

Output: To predict dem_Signal and $i_times_M(t,k,S_0,x)$

Procedure:

$[n,N,k,m,p]$ =non-zero coefficient associate with nodes N; $\forall(N) : 1 \leq N \leq n$

$Cluster_Size \leftarrow \frac{N_k}{p} \forall(k) : 1 \leq k \leq N$

MPI_Bcast(m)

for $t \in 1,2,3,...\ to\ Cluster_Size$ **do**

$x(:,m(k)) \leftarrow P_{x(:,p(k))} + S_0(:,\alpha\frac{N_n}{p})$

compute partial x with $hybrid_model$

$L_p \leftarrow \{\alpha\frac{M_p}{\Sigma_{i=1}^{k} M_i} + \beta\frac{S_p}{\Sigma_{i=1}^{k} S_i}\}$

$L_{cluster} \leftarrow \Sigma_{i=1}^{k} M_{P_i}$

end

MPI_Reduce(α)

//To find the best group of memory controllers and interconnected links

$B \leftarrow create\ buckets(n,np)$

$G \leftarrow L_c = \sum_{i=1}^{N_{pc}} W_{p_i}$

$i \leftarrow 0$

while $size(B) < np\ and\ i < size(G)$ **do**

$P \leftarrow sortedG[i]$

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

$distribute\ pairs(B, sorted\ P)$

$i \leftarrow i+1$

end

Store the aggregate result in X

Return X

accesses from different processes do not interfere with each other, thereby reducing contention and improving overall performance. In the fourth step, we apply the K-means clustering technique to determine the groups, as the group pairs of processes are initially unknown. The communication pattern requires an additional parameter, $\boldsymbol{K}$, to define the clusters in this method. This parameter significantly influences the traffic at the memory controller. Hence, estimating the number of process pairs as a group in advance is critical. The K-means clustering technique begins with $\boldsymbol{K = 2}$ and increments based on the goodness of fit test estimation algorithm. As the value of K increases, the timestamp intervals for the process pairs decrease. Ultimately, we map the communication pattern

between MPI ranks and cores within the same classified group based on identical timestamps.

These four steps are integral to the implementation of Algorithm 2 and Algorithm 3. The cache partitioning technique plays a vital role in managing and optimizing memory usage, reducing contention, and improving the overall efficiency of multi-core systems by ensuring that data accesses are efficiently handled without causing significant delays or bottlenecks.

4.3 Applying The Algorithm

4.3.1 Gathering the node information and communication pattern

In Algorithm 2, the primary objective is to reduce the load on processors through an effective cache partitioning technique. The input function, $x = i_t imes_M(t, k, S_0, x)$, is passed into the algorithm where M represents the memory bandwidth, t is the time, k is the iterative index, S_0 is a possible solution, and x denotes the performance metrics of interconnected processors. This procedure involves receiving the **dem_Signal** with the specified input parameters across all associated nodes. Although the communication overhead may increase with the number of nodes, N, it is crucial to ensure that the increment remains sub-linear. Therefore, non-zero coefficients are incorporated with associated nodes N to manage this.

The subsequent step involves determining the **Cluster_Size**, which is the ratio of the number of active nodes to the number of processors they encompass. Following this, the initial node broadcasts messages to all associated nodes using the **MPI_Bcast** routine. Execution times for all processors are then computed and partially stored in the inter-communicating nodes. This step is crucial for calculating the processor-level load based on their participation.

In the final loop, the aggregate load on the cluster is computed. The **MPI_Reduce** routine is then employed to call the load function via the load-coefficient, thereby reducing the distributed load and storing the cumulative result in a single processor. During the execution of Algorithm 2, communication patterns may vary significantly depending on the kernel's configuration. Given that the placement strategy is based on static MPI processes, it supports

optimal mapping for execution. Consequently, memory controllers and interconnected links must follow a priority list for optimization, organized from higher to lower traffic volumes.

By integrating cache partitioning techniques, this approach enhances load balancing, minimizes memory congestion, and ensures efficient use of memory bandwidth, ultimately leading to improved overall system performance.

In the next step, a weighted metric $\boldsymbol{W_p}$ containing a normal sum of messages with their size is considered. A normal sum of messages typically refers to the total count or volume of communication messages exchanged between cores or nodes within the Cluster. This measure is often used to assess the communication overhead and network traffic in parallel computing environments. the weigh metric $\boldsymbol{W_p}$ is defined by the following Eq. (4.5).

$$W_p = \alpha \frac{m_p}{\sum_{i=1}^{n} m_i} + \beta \frac{S_p}{\sum_{i=1}^{n} S_i} \qquad \text{Eq. (4.5)}$$

Where $\boldsymbol{m_p}$ is the number of messages exchanged by a pair of data (message-controller, interconnected link bandwidth), $\boldsymbol{s_p}$ is the size of messages exchanged, and $\boldsymbol{n}$ is the number of messages exchanged in all groups. The $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are the normalized values for the total number of messages and the total size of messages, respectively.

The second metric is $\boldsymbol{L_c}$, which is the load of a group $\boldsymbol{c}$:

$$L_c = \sum_{i=1}^{N_{pc}} W_{p_i} \qquad \text{Eq. (4.6)}$$

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

4.3.2 Reducing congestion with load balancing (RCLB) Algorithm

In Algorithm 3, we utilize **MPI_Bcast** to broadcast the total number of participating processors and calculate the distributed load on individual processors. This algorithm

employs C libraries to seamlessly integrate **MPI_Bcast** and **MPI_Gather** functions. After obtaining the pairs of processors from Algorithm 2, the proposed Algorithm 3 focuses on Reducing Congestion with Load Balancing (RCLB) by performing a matching between processes and their respective nodes. This matching factor ensures a balanced load across memory controllers and interconnected links, leveraging cache partitioning techniques to achieve this balance. The load balancing is facilitated by using Eq. (4.5) and Eq. (4.6).

Data distribution and processing format are critical for measuring the performance of multi-core systems, presenting a significant challenge. The proposed algorithm is also capable of evaluating the data-loading time, which is defined by Eq. (4.7). This equation calculates the load of a cluster with P processors:

$$Load_C(P) = \Sigma_{i=1}^{k} M_{p_i} \qquad \text{Eq. (4.7)}$$

where $\boldsymbol{k}$ represents the number of pairs in a group with various combinations of multi-core system components using $\boldsymbol{P}$ processors.

The RCLB algorithm distributes the load on memory controllers by creating buckets of size $\boldsymbol{n}$. These buckets are represented by nodes, and the number of cores defines their capacity. Processors are classified and placed into buckets based on their load, using a greedy approach. In this approach, no bucket contains more than one process with the same rank, and a single process cannot occupy more than one bucket at a time. Processors are executed in priority and ascending order, from smallest to largest, as outlined in lines 15 to 18 of Algorithm 3 [28, 29]. Optimization steps are mapped and placed into a single bucket without violating the aforementioned conditions, serving in a round-robin fashion. This method, enhanced by cache partitioning, ensures efficient memory usage and reduced congestion in multi-core systems.

4.4 Simulation and Result Analysis

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

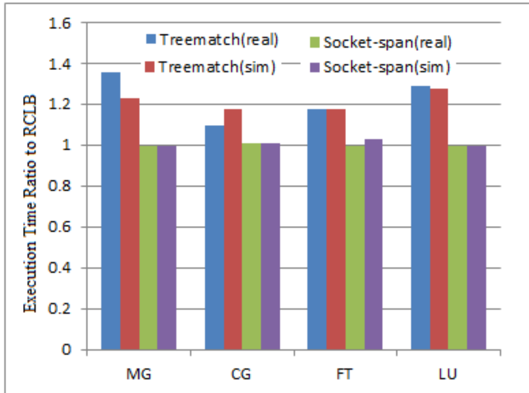

Figure 4.1: Execution time (in ms) ratio of selected kernels and proposed algorithm (3) RCLB.

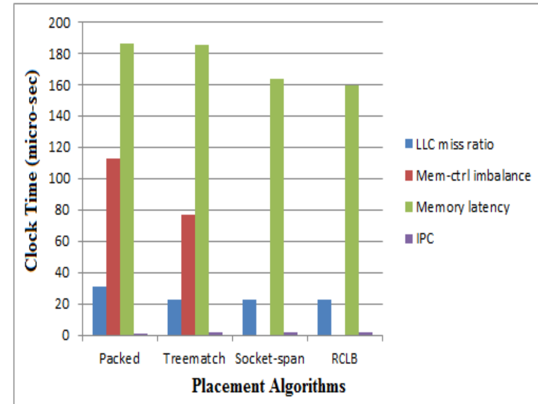

Figure 4.2: Result verification of selected kernels and proposed algorithm for memory congestion effect for MG Kernel.

- ***Based on Real system:*** The first experiment was conducted on an Intel-based Xeon-Phi-200 series-based multi-core system.
- ***Based on Virtual Cluster:*** The second experiment was conducted on three virtual clusters having 16 nodes in each.

4.4.1 Results and Evaluation

To calculate the congestion effect on different placement methods for class C and class D problems, a large-scale simulation has been conducted.

From Figure 4.1, and Figure 4.2, it is clear that the RCLB can achieve comparable performance with the method Socket-span. From the Figure 4.1, it is also clear that whenever the number of processors becomes more significant than the socket-span method, it cannot be able to maintain the memory controller imbalance. The reason is that the Socket-span method uses communication patterns of applications. We have used Simulator for Message Passing Interface (SMPI) simulator v3.20 for the experiments. This simulator can 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.

4.4.1.1 Simulation setup

To simulate the memory congestion effects on multi-core systems, SimGrid provides a virtual platform that imitates the topology of modern techniques. The resources configured on SimGrid correctly represent the cores and memory controllers. The cores interconnected with memory controller links have a higher bandwidth and lower latency. The communication between two nodes uses a memory controller link, and the interaction between two cores uses a default interconnection link.

4.4.1.2 Simulation Verification

The accuracy of our simulation results is validated by comparing them with real-time multi-core systems. As shown in Figure 4.1, the simulation outcomes illustrate system performance as a ratio of execution times between our system and others using the same method. This comparison reveals that, aside from the Fault Tolerance (FT) kernel, the socket span for both actual and simulated systems is nearly identical. The differences observed in the FT kernel can be attributed to variations in the communication patterns and memory imbalances that arise with different methods.

To address memory congestion, we utilize a cache partitioning technique that strategically allocates cache resources to balance load and improve data locality. Our platform comprises eight memory controller links and 64-Xeon-Phi-200 series processor hosts, each linked by memory controllers to hosts with identical resources. By employing Intel's memory latency checker[2], we configured the latency and bandwidth for each controller link. We intentionally reduced the bandwidth to create a high-congestion environment, maintaining the bandwidth ratio to simulate realistic execution conditions. This setup ensured significant congestion during application execution.

The environment was further configured to support OpenMP and MPI for inter-node communication. Verification results, depicted in Figure 4.1, demonstrate the effectiveness of our approach. As shown in Figure 4.2, the execution times for both actual and simulated systems are comparable, indicating that our simulations closely mirror real-world conditions. The observed differences in execution times confirm that our simulation

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

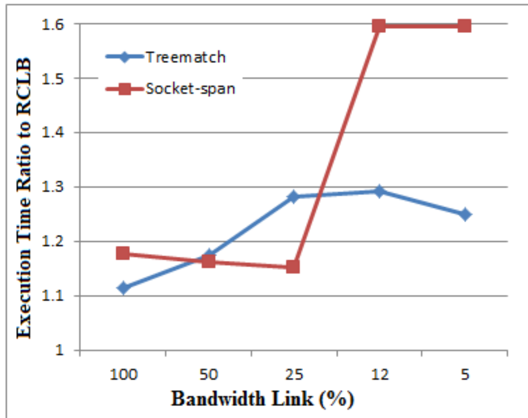

Figure 4.3: Execution time ratio of RCLB on Conjugate Gradient (CG) kernel.

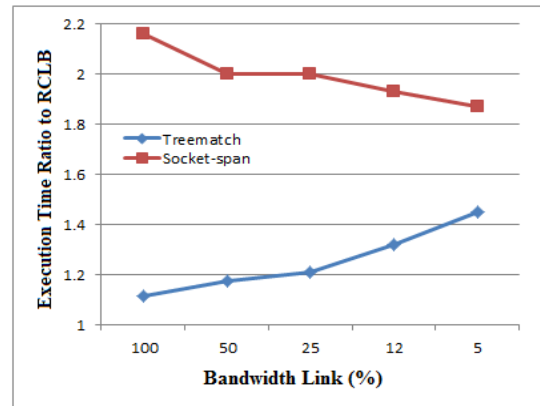

Figure 4.4: Execution time ratio of RCLB on Fault-Tolerance (FT) kernel.

accurately reflects the impact of memory congestion, validating the proposed cache partitioning technique as an effective solution for mitigating significant memory congestion issues in multi-core systems.

4.4.2 Performance comparison of simulated results

After verifying the accuracy of simulated results, the performance evaluation and analysis have been conducted on different bandwidths of memory controllers. From the other results obtained in next figures (Figure 4.7, Figure 4.8, Figure 4.9, and Figure 4.10, it is clear that Packed is the slowest method based on execution time. Thus, we skipped the packed placement method in this result. The topology of a four-node multi-core system, shown in Figure 3.1 [see Chapter 3], has been used for the simulation. The QP-Interconnect has four sockets with a memory controller and eight interconnected links in this topology. In this way, the computing node has been configured. Therefore, to use 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 the order of 50%, 25%, 12%, and 5%.

Figure 4.3, and Figure 4.4 shows the evaluated results of Conjugate Gradient (CG) kernel and Fault-Tolerance (FT) kernels for different memory congestion case. These results have been assessed on hwloc tracing and analyzing features. These features also allow finding

and analyzing the latencies and utilization.

Figure 4.3 demonstrates the performance of the proposed method concerning CG kernel. The proposed method shows a better result based on increased Bandwidth (%) in this figure. As mentioned earlier, the communication pattern of the CG kernel is irregular, so the socket-span method shows better results than other methods. Since the Socket-span suffers from high latencies of slowly communicating processes, without considering the communication 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 the memory controller becomes 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 4.4 presents the performance results of the proposed method on FT kernels. This result clearly shows that the Socket-span needs two times longer execution time than the proposed method. The FT kernel operates all to all communication patterns. This is the reason why Socket-span sufferers from higher latencies. In the case of Treematch, this method reduces communication locality by reducing remote access. Similarly, the proposed method reduces the number of remote accesses by considering communication locality. The only difference between Treematch and the proposed method execution time is that as the Bandwidth decreases, the performance increases. Thus, considering these results, we can say that the proposed method can reduce memory congestion.

Similar patterns of execution time ratio with the proposed method have been shown in Figure 4.5 and Figure 4.6. Figure 4.5 shows the performance of the proposed method on the Lower Upper (LU) kernel, and (b) shows the performance results of the proposed method on the Multi-Grid (MG) kernel. Earlier sections mentioned that the MG kernel works on irregular communication patterns, and the LU kernel does the same. The traffic on the MG kernel is larger than the LU kernel. So, the Socket-span suffers from higher latency again, and execution time increases between all communicating nodes. As a result, the congestion level becomes too low; hence, the load imbalance among all memory controllers is also reduced. Here in Figure 4.5 and Figure 4.6, the execution time of Treematch and the proposed method is almost the same, and they are achieving better execution time than Socket-span.

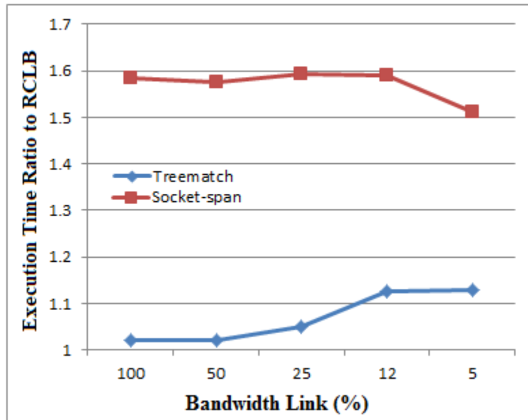

Figure 4.5: Execution time ratio of RCLB on Lower Upper (LU) factorization Kernel

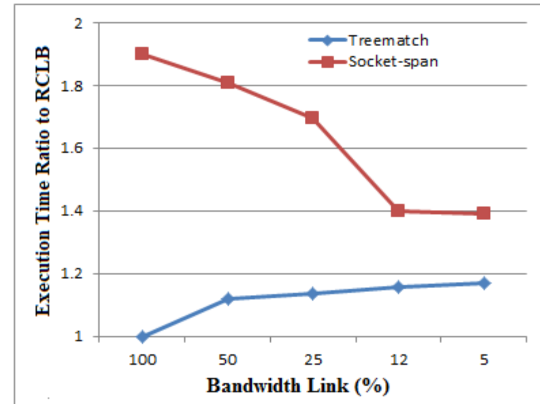

Figure 4.6: Execution time ratio of RCLB on Multi-Grid (MG) Kernel

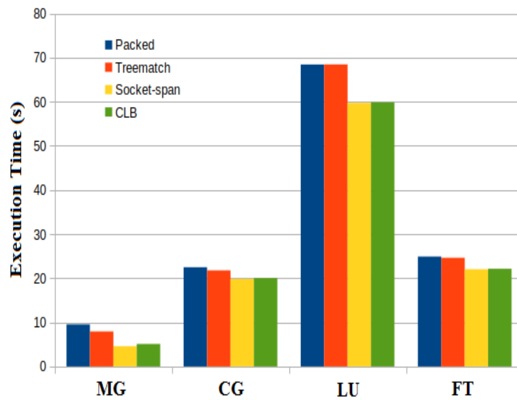

Figure 4.7: Average execution time(in sec) of the NPB kernels for class C problems

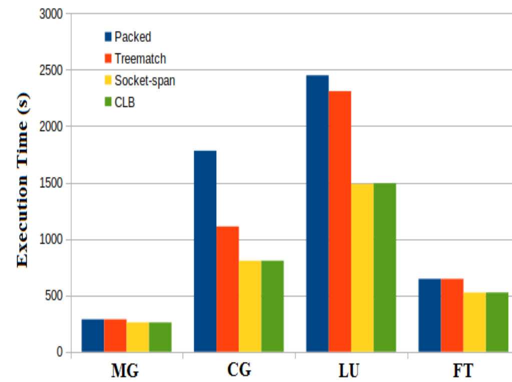

Figure 4.8: Average execution time (in sec) of the NPB kernels for class D problems

4.4.3 Performance comparison of NPB kernels

Figure 4.7 and Figure 4.8 illustrate the average execution times for Class C and D of NPB kernels, representing moderate and large problem sizes. We conducted 50 sample executions to compute the average execution times for different seeds, with the mean values calculated at a 97.33% confidence level. This high confidence level indicates that the average execution times shown in Figure 4.7 and Figure 4.8 closely approximate the sample means.

In Figure 4.7, the RCLB technique demonstrates superior performance compared to the

stock-span, which in turn surpasses both the packed and tree-match techniques. The stock-span method generally outperforms the packed, Treematch, and RCLB methods by reducing the number of accessed communications. However, it still involves some remote-access operations that require substantial computational effort [102]. Treematch, on the other hand, attempts to assign communications to the same node by increasing the load on the memory controllers of that node, leading to imbalanced memory controllers and increased total execution time.

RCLB effectively mitigates memory congestion by implementing a load balancing policy, which is why it performs better than the packed and Treematch techniques. Although the stock-span method performs well across all kernels, it does not consider the communication patterns of the cores, leading to a suboptimal distribution of data among nodes. As a result, stock-span reduces the load balance between two communicating nodes without optimizing for communication patterns. In contrast, RCLB uses cache partitioning techniques to ensure that communication patterns are taken into account, which helps in evenly distributing the data and preventing memory congestion. This thoughtful consideration of communication patterns makes RCLB a more effective solution for reducing memory congestion in multi-core systems.

Table 4.1: 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

Given that the RCLB and Socket-span techniques exhibit nearly identical average execution times, they can be considered comparable, with similar mapping results across all nodes. Table 4.1 presents the development of four placement algorithms for the MG kernel. In this table, each MPI process is represented by its rank number, and cores are denoted by ID numbers ranging from 0 to 15. While the default core mapping differs from those of Socket-span and RCLB, their underlying mapping techniques are nearly identical. Socket-span allocates MPI processes by searching for neighboring processes across nodes, arranging them in a binary sequence: (0, 1, 0, 1). The RCLB method also assigns cores and nodes in a binary format, but with a different sequence: (0, 1, 0, 1, 1, 0, 1, 0). These

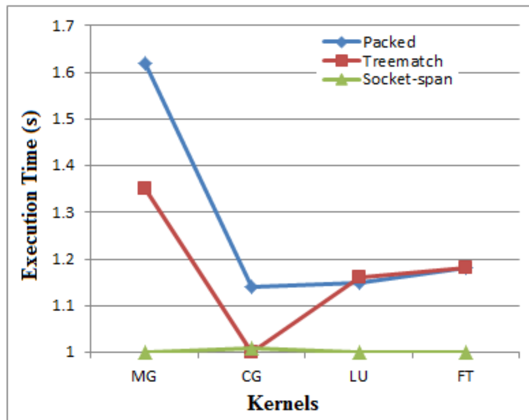

Figure 4.9: Normalized execution times of the NPB kernels for class C problems

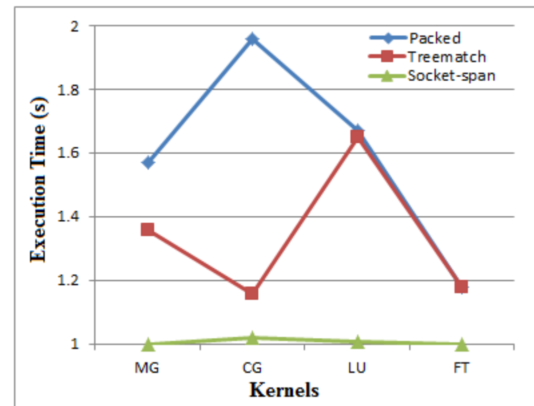

Figure 4.10: Normalized execution times of the NPB kernels for class D problems

sequences correspond to different placement methods. Ultimately, the results indicate that RCLB and Socket-span produce nearly identical outcomes.

For large problem sizes, such as those using NPB Class D, the results shown in Figure 4.8 demonstrate that Socket-span and RCLB outperform other techniques for all four kernels. The MG kernel exhibits the minimum performance, while the LU kernel shows the maximum. The lower performance for the MG kernel suggests minimal data exchange between interconnected nodes, whereas the higher performance for the LU kernel indicates substantial data exchanges. Both RCLB and Socket-span show comparable performance across all benchmarks, including MG, CG, LU, and FT [31], implying that performance gains are maintained even for large problems. In these scenarios, the communication pattern follows an all-to-all approach, minimizing the impact of remote access communication on overall performance.

The performance similarities between RCLB and Socket-span can be attributed to their effective use of cache partitioning techniques. By strategically partitioning the cache, both methods optimize data locality and reduce memory congestion, thereby enhancing execution efficiency. This cache partitioning ensures that data frequently accessed by MPI processes is kept close to the cores, minimizing latency and improving overall system performance. Thus, the integration of cache partitioning with RCLB and Socket-span plays a crucial role in achieving balanced and efficient memory utilization across multi-core systems.

Figure 4.9 and Figure 4.10 depict the normalized execution times for class C and class D problem sizes, respectively. In these figures, the RCLB technique serves as the baseline for comparison, with all other methods being derived from it. Consequently, the results are benchmarked against the performance of RCLB. From Figure 4.9, it is evident that each method requires more time than the baseline. Notably, the Socket-span method demonstrates a lower or equal execution time compared to RCLB, indicating its potential for improved performance. However, as the problem size increases, both the Packed and Treematch methods exhibit increased execution times. This increase is attributed to the growing load imbalance among nodes, which these methods fail to mitigate effectively. Hence, RCLB proves superior in reducing load imbalance across interconnected nodes, demonstrating better performance even under significant memory congestion conditions, as illustrated in Figure 4.9 and Figure 4.10.

In Figure 4.1, the MG kernel's performance is evaluated using the Linux Perf-profiling tool to monitor memory controller activities, complemented by an Intel-based profiling tool. Both tools reveal that memory controller imbalance is minimal for the Socket-span and RCLB methods, while other placement methods show significant imbalances. These disparities in performance are primarily due to differences in memory congestion. Figure 4.2 presents the performance variations of each placement method, using four benchmark metrics: LLC miss ratio, memory controller imbalance, memory latency, and IPC. The Packed method exhibits the highest LLC miss ratio, indicating the highest memory congestion risk, while the Socket-span method shows the lowest ratio. Notably, the LLC miss ratio for Socket-span and RCLB is nearly identical, suggesting a promising area for further exploration. Higher execution times correlate with increased memory latency due to prolonged CPU waits for LLC miss fulfillment. A higher IPC indicates better performance for the respective placement methods, as demonstrated in Figure 4.2, validating the results obtained for the MG kernel. The conclusions drawn from Figure 4.2 are based on three key observations:

1. The Packed method has the highest LLC miss ratio and longest memory latency due to its lowest IPC.
2. RCLB and Socket-span methods exhibit lower memory latencies and higher IPCs compared to other placement methods, highlighting the impact of load imbalance on memory latency.

3. RCLB and Socket-span effectively reduce memory congestion by preventing load imbalance in memory controllers.

4.4.4 Performance measure based on locality and congestion

To assess multi-core system performance, benchmarks offer valuable insights. NAS benchmarks are widely recognized and accepted for this purpose. Among these, the Princeton Application Repository for Shared-Memory Computers (PARSEC) stands out as an exemplary representation of shared memory architecture, particularly effective for studying NUMA (Non-Uniform Memory Access) effects due to its utilization of over 30% of CPU resources. Our experiments were conducted on an AMD server equipped with four quad-core CPUs.

The initial experiment aimed to understand the impact of memory congestion on basic systems. This involved running benchmarks that imposed limited stress on memory controllers and interconnect links via a single thread. The experiment compared remote and local configurations. Local configurations featured memory and threads operating on the same node, while remote configurations shared memory across different nodes using a standard Linux tool.

Cache partitioning techniques were pivotal in these experiments. By strategically dividing the cache among different cores, we aimed to enhance memory access efficiency and reduce contention. This approach was critical in distinguishing the performance outcomes between local and remote configurations.

Table 4.2: Traffic congestion effects over selected approaches (Packed and Socket-span)

	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

The distinctions between Packed (P) and Socket-span (S) configurations are highlighted in Table 4.2. This table presents metrics derived from multi-threaded scenarios. It compares the performance of two greedy approaches: Packed and Socket-span, demonstrating absolute performance across various metrics. The table elucidates the performance disparity between the best and worst policies for the benchmarks used. Notably, the percentage of memory-controller imbalance is minimized across all applications with the optimal policy. These findings underscore the significance of cache partitioning in optimizing multi-core system performance, mitigating memory congestion, and enhancing overall efficiency.

4.5 Summary

In this chapter, we introduce the Reconciled Locality for Memory Congestion Reduction (RCLB) method aimed at alleviating memory congestion through optimized data locality. By addressing load imbalances on memory controllers and leveraging MPI communication patterns within clusters, RCLB effectively manages data traffic flow. Evaluation against two greedy approaches, namely packed and stack-span, demonstrates RCLB's superior performance. Additionally, we propose a novel architecture model to classify multi-core characteristics at the hardware level, tailored for our unique architecture. Our algorithm showcases the ability to fully exploit the maximum capacity of multi-cores, a quantitative advancement in mitigating memory congestion. Despite hardware advancements, our findings underscore the persistent bottleneck in system performance. Real-world simulations confirm RCLB's effectiveness, showing significant bandwidth enhancements ranging from 12.65% to 23% in memory-intensive tests.