Exploiting Programmable Network Interfaces for Parallel Query Execution in Workstation Clusters

V. Santhosh Kumar[1], **M. J. Thazhuthaveetil**[2], **and R. Govindarajan**[2]

[1]Supercomputer Edn. and Res. Centre
Indian Institute of Science
Bangalore 560 012, India

gvsk@hpc.serc.iisc.ernet.in

[2]Supercomputer Edn. and Res. Centre
Dept. of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012, India
{mjt,govind}@serc.iisc.ernet.in

Abstract

Workstation clusters equipped with high performance interconnect having programmable network processors facilitate interesting opportunities to enhance the performance of parallel application run on them. In this paper, we propose schemes where certain application level processing in parallel database query execution is performed on the network processor. We evaluate the performance of TPC-H queries executing on a high end cluster where all tuple processing is done on the host processor, using a timed Petri net model, and find that tuple processing costs on the host processor dominate the execution time. These results are validated using a small cluster.

We therefore propose 4 schemes where certain tuple processing activity is offloaded to the network processor. The first 2 schemes offload the tuple splitting activity – computation to identify the node on which to process the tuples, resulting in an execution time speedup of 1.09 relative to the base scheme, but with I/O bus becoming the bottleneck resource. In the 3rd scheme in addition to offloading tuple processing activity, the disk and network interface are combined to avoid the I/O bus bottleneck, which results in speedups upto 1.16, but with high host processor utilization. Our 4th scheme where the network processor also performs a part of join operation along with the host processor, gives a speedup of 1.47 along with balanced system resource utilizations. Further we observe that the proposed schemes perform equally well even in a scaled architecture i.e., when the number of processors is increased from 2 to 64.

1 Introduction

Cluster computer systems, assembled from commodity off-the-shelf components, have emerged as a viable alternative to high-end custom parallel computer systems for applications demanding high performance [1]. An important component in such cluster computer systems is their high performance interconnect with programmable network interfaces such as Myrinet, Quadrics, and Infiniband [10].

The network processors available in such programmable interfaces often have the capability to perform certain application related processing, thus facilitating interesting opportunities to enhance application performance in cluster systems.

In this paper we consider parallel database query execution on such a cluster. Commercial database and data mining applications, like scientific applications, are highly parallel and well suited for parallel computers [5]. They have been implemented on a variety of parallel systems, including cluster computers [5, 20]. However, unlike most scientific applications, in database applications disk I/O costs are a dominant factor. So, most efforts at improving database application performance concentrate on reducing the effective disk I/O cost [6]. Further developments in fast RAID technology equipped with high bandwidth disk and large main memories may render disk I/O as a less dominant resource in query execution. Hence, in this paper, we conduct a detailed performance evaluation of parallel query processing on workstation clusters having high performance disks with high disk bandwidth and larger memories.

We have developed a timed Petri net model for the parallel execution of database query on a cluster. We study the performance of queries from the TPC-H benchmark [21]. A salient feature of our Petri net model is that it captures the application, the architecture, and their interaction. Thus the model for the execution of each query is different. We validate our Petri net model against the actual execution of hash join on cluster of 4 machines connected by Myrinet NIs and an 8-port switch.

First we study the performance of a base scheme where all the application related tuple processing is performed by the host processor (HP). In this system, despite using a high performance contemporary processor in the cluster, the host processor is the bottleneck, while other resources such as the network processor (NP) have low utilization. We start by offloading the tuple splitting task from the host processor (HP) to the network processor (NP). Our first two suggestions are software schemes, *Network Interface Tuple Splitting (NITS)* scheme, where tuple splitting is done by NP only, and *Duplicate Tuple Splitting (DTS)* scheme, where tuple splitting is done by both HP and NP. In these schemes, the I/O bus becomes the system bottleneck, limiting the performance improvement. We therefore propose an architectural enhancement where the disk is directly attached to the NI, with NP performing the tuple splitting. The resulting scheme, referred to as *Network Interface with Attached Disk (NID)*, yields improved performance; but the HP once again becomes a resource with high utilization. We finally propose a *NI Join (NIJ)* scheme where a part of the tuple processing activity is offloaded to the NP. This scheme results in a significant improvement in execution time speedup of 1.47 relative to the Base scheme with reasonably balanced resource utilizations. More importantly, the performance improvement increases with increase in the computing power of NP. These results suggest how higher processor power in future network interfaces can be effectively utilized by applications. We performed two types of scalability tests on the proposed schemes in large clusters. In the first, the problem is scaled as the cluster size is increased; in the second, the problem size is kept constant while increasing the cluster size. We found that in both tests, the proposed schemes exhibit near linear speedup, which is an important criterion for the usefulness of the schemes.

The main contributions of this paper include:

- A detailed Petri net model of parallel query execution on a cluster of workstations, where the application behavior, architecture, and their interactions are modeled.
- Software and hardware schemes which take advantage of programmable network processors, to improve the performance of parallel query execution on workstation clusters.

The rest of the paper is organized as follows: In Section 2, we briefly review the background on clusters, programmable network interfaces, and query processing. We describe our Petri net model for the *Base* scheme in Section 3. This section also reports the performance of the *Base* scheme. Section 4 discusses the *Network Interface Tuple Splitting (NITS)* scheme and its simulation results. Following this in Section 5 we discuss the *Duplicate Tuple Splitting (DTS)* scheme and its performance. In Section 6, we describe the *Network Interface with attached Disk (NID)* scheme and its simulation results. We describe our *Network Interface Join (NIJ)* scheme and its results in Section 7. Finally scalability results are reported in Section 8. Related work is discussed in Section 9 and concluding remarks are provided in Section 10.

2 Background

2.1 Clusters

A cluster of workstations is a distributed memory machine where each node is a stand alone system with CPU, memory attached to the memory bus, and peripherals like disk, and network interface attached to the I/O bus. We assume such a cluster of N nodes. Typical fast cluster interconnects like the Myrinet network interface (NI) [13] found in current day systems have NIs with a programmable network processor (NP), on board memory (SRAM), a host DMA engine (HDMA), an EBUS Interface (64-bit), a send DMA engine (SDMA) and a receive DMA engine (RDMA). The HDMA is used to transfer data across the I/O bus to the node memory. SDMA and RDMA are used to transfer data from the NI SRAM to the communication network (switch), and vice-versa. We assume that the switch employs wormhole routing to transfer packets between network interfaces.

Research in communication layers for high performance scientific applications has lead to the development of user-level communication techniques which have reduced the involvement of HP in communication to deliver better application performance [19]. We, therefore, assume such a user-level communication layer in our system [7].

2.2 Parallel Query Processing

In a relational database system relational queries composed of relational operators like select and join are used to manipulate data. The join operator, which combines tuples from two relations based on a common attribute, is the most crucial and expensive operator [12]. Previous research has shown that hash-based join algorithms are more efficient than other join algorithms, such as sort-merge or nested-loop, in systems with large main memories [12]. So, in this work, join operators in queries are executed using hash-join algorithms. We assume that each node has enough main memory so that the hash table fits into the main memory and no extra disk I/O is required other than for reading the relations. Since complete query execution is being modeled, we also consider the select operator. Although the join operation can also be executed efficiently when indexes are present, we do not consider indexes here [20].

Since query processing is a highly parallel resource intensive task, several parallel query processing techniques have been devised and employed in parallel database machines to improve query execution time [5]. Since a cluster of workstations is essentially a shared-nothing architecture in which intra-operator parallelism is better exploited [5, 16] with horizontally partitioned relations, in our work we consider only exploiting intra-operator parallelism and horizontally partitioned relations.

When a query involves multiple joins, a query tree or a query execution plan is used to represent the scheduling sequence of the constituent operations. Query trees are characterized as left-deep, right-deep or bushy trees. Right-deep and bushy trees have multiple operators simultaneously active, and are suitable for pipelined implementations in multiprocessor systems [17, 4]. Left-deep trees assume that one operator is simultaneously active on all the nodes; the next operator starts after the current one is completed. Thus a left-deep tree based query execution plan can easily be mapped to intra-operator parallelism, and is well suited to shared-nothing architectures. We adopt a left-deep query tree representation for the queries we modeled.

The parallel hash join used in our study works as follows. It involves two phases, namely (i) the Build Phase, where the inner relation is hashed on the join attribute and a hash table is built, (ii) the Probe Phase, where the outer relation is hashed on the join attribute using the same hash function used in the build phase, and tuples are generated on successful matches. In the parallel version, a separate hash function is first used to determine the cluster node where each tuple will be processed. We refer to this activity as tuple splitting. The tuples are routed to the respective cluster node. The join operation (build and probe) takes places in that node. Thus both phases of parallel join involve communication between cluster nodes. In case a select or project operation is required it is first applied on the tuples and then the join operation takes place.

3 Base Scheme

Our objective is to study the performance of parallel query execution on a cluster and evaluate the potential of offloading various amount of tuple processing from the host processor (HP) to the network processor (NP). We develop a Petri net model for the query execution on a cluster. Petri nets can model properties like concurrency, synchronization, conflict etc. As our model has to capture timing of query processing activities and probabilities associated with tuple distribution, we used a stochastic timed Petri net. In our Base scheme, query execution proceeds in a conventional way, with the query processing done on HP and message sends and receives done by NP.

3.1 Base Scheme Model Description

Figure 1 shows our Petri net model for a single node of a cluster performing parallel query execution. For clarity, only disk accesses and join operations are shown in the figure, while the text describes the complete model. In the figure, thin lines represent instantaneous transitions and thick rectangular bars represent timed transitions. We use the name of a timed transition *e.g.,* T_DiskRd to represent the duration of the timed transition. The number of disk blocks constituting the relation to be processed is modeled by a fixed number of tokens in place $P\_DskBlkCnt$ (initial marking). With the availability of HP (modeled by a token in place P_HProc), transition $T\_InitDskRd$ fires and a request to read a disk block is placed in P_DskReq. When the disk is free, modeled by the availability of a token in place P_Disk, a disk block containing a fixed number of tuples is read from the disk. The disk read is modeled by the timed transition T_DiskRd. The tuples are transferred to the host memory ($P\_DskTupAv$) on the availability of I/O bus (P_IOBus), and the transfer time is $T\_IOBusDsk$.

HP then computes the $node\_id$ to which the tuples are to be routed. This is modeled by the transition T_Split, which places the tuples into P_TupGrp. The model assumes that tuples are uniformly distributed across the nodes of the cluster. Tuples that are to be processed on the same node are placed into $P\_TupBuff$. Depending on whether a Build Phase or Probe Phase is currently under progress, determined by the non-availability or availability of tokens in $P\_ProbeEn$, tuples either go through the Build Phase (firing of T_Build) or Probe Phase (firing of T_Probe). In the Probe Phase, depending on the join selectivity (denoted by $JoinSelProb$), tuples which qualify in the join process fire T_TupSel and those that are dropped fire T_TupDrp. For tuples destined for other nodes (T_Other), HP groups the tuples into messages and initiates a message send operation by enqueuing the message to the network interface. T_Move represents the time required for tuple grouping[1] and buffer management and T_Send the software overhead for initiating a send operation.

The places and transition used in the following description of the Petri net are not shown in Figure 1. Depending on the availability of the I/O Bus, NP initiates the HDMA engine to transfer the message from host memory to SRAM on the network interface. This is modeled by the timed transition $T\_IOBusNI$. The SDMA engine then puts the message on the network link to the input port of the switch, taking T_SDMA time units. T_route represents the time required to route the message from the input port to the switch output port where the destination node is attached.

[1] Grouping of tuples which are destined for the same remote node form a single message.This avoids the overhead of sending several small messages. We group upto 128 tuples in a single message.

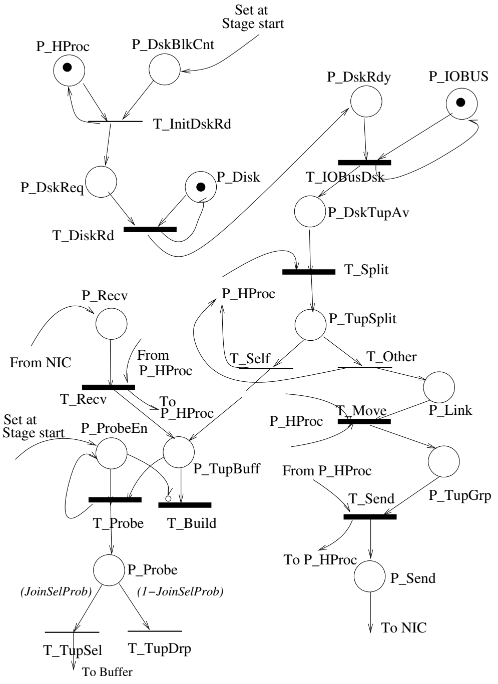

Figure 1. Petri net model for Join operations

The incoming messages at the network link are transferred to the SRAM of NI when the RDMA engine is available, and from there to the host memory, when both the HDMA and the I/O bus are available. These transitions and their durations are modeled by T_RDMA and $T\_IOBusNI$ respectively. The software overhead incurred by HP in receiving messages is T_RecvHP. The received tuples are then transferred to $P\_TupBuff$ for join processing.

When all tuples of the relation have been processed, synchronization messages are exchanged between nodes to signify the end of a phase (build or probe phase). This synchronization ensures that only intra-operator parallelism is exploited in our model. The number of tuples to be processed during the next operation is then loaded, and the process continues for other operations in the query.

The N node cluster can be modeled using a colored Petri net. However, since the traffic generated by a single node is uniformly distributed and sent to other nodes in the cluster, we can model the traffic originating from other nodes to this node by routing the messages generated from the single node back to itself through the switch. Thus a N node cluster can be modeled using the Petri net model for a single node with the above changes. We found that the difference in results between the full simulation of a N node cluster and a single node simulation is within a small percentage ($<$ 0.1%). A similar modeling approach was used by Govindarajan, *et al* [8] in developing Petri net models of multithreaded multiprocessor architectures.

3.2 Model Parameters

The architectural parameters of our model are set to represent contemporary high performance computing nodes. Since our objective is to study the effect of offloading tuple processing activities done by HP to NP, we model these in detail. We estimated the time taken for such tasks for a 3.6GHz system (using measurements on an available Pentium 4 system). The parameters for host communication overheads were obtained from measurements on Myrinet user-level messaging software running on Myrinet LANai 7 processor [13]. Network Interface parameter values are set assuming a NI SRAM bandwidth of 1.6GB/s (200MHz, 64 bit bus). The network link bandwidth is assumed to be 4GB/s based on the Myrinet specifications [18].

The I/O bus transfer time parameters are set assuming 1GB/s (64 Bit, 133 MHz) bandwidth, similar to that of PCI-X bus [14]. Disk system parameters are set assuming the use of high performance disks and capability to satisfy sequential reads at sustained high bandwidth. The disk I/O time, is set assuming the ability to deliver 1.28GB/s (4 * 320MB/s SCSI disk) in 64KB chunks. The values of the various architectural parameters are shown in Table 1.

We set the database related parameters based on the TPC-H benchmark [21]. TPC-H queries 3,5,7-10 which are join intensive (upto 4 joins) were modeled, each using a separate Petri net model, as the number of relations involved and database parameters and the operation involved are different for different queries. We assume a tuple size of 128B for all relations. The number of tuples per table is set so that the horizontal partition of the relation in each node occupies 1GB. Further, we assume that there is no skew in the data, *i.e.,* the frequency of all key values used in the join attributes occurs with equal frequency and hence tuples are uniformly distributed across the nodes. When a join is performed, a part of the tuple is projected out. The size of the projected tuple is 32 bytes. The model parameter values for selectivity ratios were obtained using measurements from query execution on a single node running PostgreSQL. The query parameters are shown in Table 2. Si's represent the selectivities of select operation and Ji's represent the join probabilities.

3.3 Performance of Base Scheme

We simulated our Petri net models using CNET, an event-driven Petri net simulation tool [22]. The simulator reports the total simulation time for the Petri net model, as well as the total firings times for each timed transition. For discussing the performance of different schemes, we use simulation results for a 8 node cluster. We have also studied the scalability of the schemes (up to 64 nodes) which are discussed in Section 8. Results reported are averaged

Parameter	Duration for	Parameter Value (μs)
	Host Processor	
$T\_SplitHP$	Split Hash function	0.00027
$T\_InsertHP$	Insert into hash table	0.17
$T\_ProbeHP$	Probe hash table	0.30
T_MoveHP	Moving tuple attributes	0.50
T_SendHP	User level Send	0.40
T_RecvHP	User level Receive	0.18
	Network Interface	
T_EnDMA	Enable DMA Engine	0.01
T_SDMA	SDMA (NI to Network)	1.28
T_RDMA	RDMA (Network to NI)	1.28
T_SW	Switch Delay	8.19
	IO Bus	
$T\_IOBusDsk$	IO Bus (Disk to Host)	61.59
$T\_IOBusNI$	IO Bus (Host form/to NI)	3.84
	Disk	
T_DiskRd	reading 64KB Block	48.82

Table 1. Simulation Parameters: HP-1X Configuration

Query 3	S1(R1) J1 S2(R2) J2 S3(R3)

Selectivity
Join Selectivity
Relation Size
	S1=0.2, S2=0.48, S3=0.54
	J1=0.10, J2=0.99
	R1=150016, R2=1499648, R3=6001152

Query 5	S1(R1) J1 R2 J2 R3 J3 R4

Selectivity
Join Selectivity
Relation Size
	S1=0.15
	J1=0.038, J2=0.577, J3=0.971
	R1=1499648, R2=6001152, R3=149504
	R4=9728

Query 7	R1 J1 S1(R2) J2 R3 J3 R4

Selectivity
Join Selectivity
Relation Size
	S1=0.30
	J1=0.006, J2=0.33, J3=0.65
	R1=6001152, R2=1499648, R3=9728
	R4=149504

Query 8	S1(R1) J1 R2 J2 R3 J3 S2(R4) J4 R5

Selectivity
Join Selectivity
Relation Size
	S1=0.007, S2=0.30
	J1=0.0002, J2=0.43, J3=0.029, J4=0.09
	R1=199680, R2=6001152, R3=9728
	R4=1499648, R5=149504

Query 9	S1(R1) J1 R2 J2 R3 J3 R4 J4 R5

Selectivity
Join Selectivity
Relation Size
	S1=0.06
	J1=0.06, J2=0.99, J3=0.06, J4=0.25
	R1=199680, R2=799744, R3=9728
	R4=6001152, R5=1499648

Query 10	S1(R1) J1 S2(R2) J2 R3

Selectivity
Join Selectivity
Relation Size
	S1=0.04, S2=0.24
	J1=0.15, J2=0.38
	R1=1499648, R2=6001152, R3=149504

Table 2. Query Parameters

across 5 independent runs for each query.

We use relative speedups of query execution times for performance comparison and the utilization of resources like host processor (HP), Disk, I/O Bus, Switch (SW) and network processor (NP) to identify bottleneck resources. In discussing the results, we report the average of the execution time of all the queries for a given scheme. The relative speedup of a scheme is the ratio of the average query execution time in the Base scheme (with HP-1X parameters in Table 1) to that in the proposed scheme. Resource utilization is computed as the ratio of total firing time of the transition involving the resource to the total simulation time of the query.

Table 3 shows the results for the Base scheme using the HP-1X parameter values. Observe from Table 3 that the query execution time is dominated by the tuple processing cost of HP, which has a utilization of 92.9%. We also consider another parameter setting where the HP speed is doubled (*i.e.,* the HP parameter values in Table 1 halved), which we refer to as HP-2X. Observe that doubling HP power yielded a relative speedup of 1.63 with respect to HP-1X configuration, showing that tuple processing activities done by HP are a significant factor in query execution time. The other resources I/O Bus, Disk, Switch and NP are not bottleneck resources with utilizations of 54.9%, 32.3%, 15.8% and 0.1% respectively. The high resource utilization of HP (92.9%) and low utilization of NP (0.1%) motivated us to study options of offloading work from HP to the programmable processor in the network interface.

HP	Utilization (%)					Exec Time(s)	Relative Speedup
	Disk	HP	I/O Bus	SW	NP		
1X	32.3	92.9	54.9	15.8	0.1	2.13	1.00
2X	45.3	63.4	78.2	22.0	0.1	1.31	1.63

Table 3. Resource Utilization and Speedup for Base Scheme

3.4 Petri net model Validation

Before proceeding further, we validate our Petri net model against a uniprocessor implementation of hash-join (1 node) and with MPI based implementations on 2 nodes and 4 nodes. The implementation of hash join was carried out on 3.4GHz Pentium 4 machines with 1GB RAM. The Myrinet Network Interface card uses a 134MHz LANai 9.2 processor seated on 32bits/33MHz PCI slots. The machines were interconnected with a DUAL-8-PORT Myrinet switch working at 1.28 Gbits/s per link, dedicated for MPI communication.

	Execution Time(s)		
	1 Node	2 Node	4 Node
Actual Execution Time	0.22	0.41	0.51
Petri net Simulation Time	0.21	0.36	0.43

Table 4. Validation of Petri net model

In our validation, we used two in-memory relations with 512K tuples per node for each relation. We assume a join probability of 0.5. The keys were chosen from a uniformly distributed random variable. Table 4 shows the performance results for actual execution time and Petri net simulation time. The problem size (total number of tuples) is scaled as the number of processors is increased. We find that the difference in execution times is less than 16%. This difference is partly due to the MPI_test overhead incurred by a non-blocking receive, which is not explicitly modelled in our Petri net model.

4 Network Interface Tuple Splitting (NITS) Scheme

During a parallel join under our Base scheme, the host processor (1) builds and probes the hash table, (2) performs tuple splitting.

In our NITS scheme, we propose offloading the tuple splitting activities to NP. This is achieved as follows: Relations available on the disk are read as blocks into the main memory and made available in contiguous buffers which can be DMAed. This enables NP to transfer a relation from main memory to the SRAM in the NI and to take responsibility for tuple splitting and grouping activities involved in distributing these tuples across the cluster nodes. More specifically, tuples which are to be processed locally are DMAed back to HP. The rest are communicated to their respective destinations as in the Base scheme. Thus in the NITS scheme, HP is relieved of the tuple splitting activities and works only on the hash join operation.

The Petri net model of the Base scheme was modified to account for the tuple processing activities being done by NP. The execution time for the tuple splitting tasks depend on the NP processing power. Typical NP clock speed is much lower than that of the associated HP. For instance, a contemporary HP runs at 3.6GHz while the Myrinet LANai 2XP clock is 300Mhz, which is one twelfth of the HP clock. Technology trends suggest [18], that NP clock rates are increasing. To study the sensitivity of the performance of our schemes to NP computing power, we added a model parameter *HP:NP*, the ratio of the processing powers of HP to NP.

The relative speedup for queries and resource utilizations under the NITS scheme are compared with those for the Base scheme in Table 5. We see that there is reduction in HP utilization from 92.9% for the Base scheme to 76.1% for the NITS (*HP:NP* $=$ 1:1) scheme, and an increase in the nework processor utilization (11.6%). Although work has been offloaded from HP, the bottleneck resource in the Base scheme, we find the relative speedup to be less than 1 for all values of *HP:NP*.

The data in Table 5 reveals why this happens. Observe that with *HP:NP* ratios of 1:1/9 and 1:1/8, that NP is the highest utilized resource (with utilization $>$ 73.4%) suggesting that the computing power of NP is insufficient to handle the tuple splitting activities, and thus limits the performance gains. With increase in NP power (*HP:NP* ratios ranging from 1:1/7 to 1:1/6), the speedup increases from 0.83 to 0.89 and the I/O bus becomes the highest utilized resource (utilization ranging from 77.3% – 84.6%). This suggests that though there is some performance benefit from increasing NP power, the I/O bus has become the bottleneck. Increasing NP computing power further (from 1:1/4 through 1:1) results in no improvement in speedups, but worsens the I/O bus utilization, confirming that the I/O bus has become the system bottleneck.

Scheme	HP:NP	Relative Speedup	Utilization (%)				
			Disk	HP	I/O Bus	SW	NP
Base		1.00	32.3	92.9	54.9	15.8	0.1
NITS	1:1/9	0.72	23.5	59.6	66.2	11.5	76.6
	1:1/8	0.77	25.3	64.2	71.3	12.4	73.4
	1:1/7	0.83	27.4	69.6	77.3	13.4	69.7
	1:1/6	0.89	29.9	75.9	84.3	14.6	65.3
	1:1/4	0.90	30.0	76.0	84.5	14.7	43.9
	1:1/2	0.90	30.0	76.0	84.5	14.7	22.3
	1:1	0.90	30.0	76.1	84.6	14.7	11.6

Table 5. Comparison of Relative Speed-Up and Resource Utilization: Base and NITS schemes

We therefore studied the effects of doubling I/O bandwidth under the NITS scheme. We observed an increase in speedup (0.89 to 1.12), by doubling I/O bus bandwidth. We looked into what caused the I/O bus to become the bottleneck and found that it was the traffic generated by tuples being DMAed over the I/O bus, along with the traffic caused by tuples being routed from NP to HP to be joined locally on the node. Our next attempt, therefore, is to look at modifications which eliminate the latter traffic.

5 Duplicate Tuple Splitting (DTS) Scheme

In this scheme we duplicate the task of splitting of tuples on both the host and the network processor, and hence the name Duplicate Tuple Splitting (DTS). All tuples read off the disk are examined by both HP and NP. The HP computes $node\_id$ for all the tuples and processes (probes) only those destined for itself, ignoring the remaining tuples. The NP computes $node\_id$ for all the tuples after performing DMA operation of tuples in suitable chunks into NI SRAM. It then ignores tuples to be processed locally and communicates the rest to other nodes based on their $node\_id$. This modification to the NITS scheme avoids the transfer of tuples to and fro on the I/O bus, to be joined locally.

The speedup and resource utilizations under the DTS scheme are compared with that of the Base scheme in Table 6. We see that when *HP:NP* is 1:1/9 or 1:1/8, the speedup is less than 1, and that NP has a high utilization of 71.4% and suggesting that NP's computing power is not sufficient to perform the tuple splitting task. However, with *HP:NP* $=$ 1:1/7 and *HP:NP* $=$ 1:1/6 the speedup increases to 1.07, and 1.09 respectively. NP is no longer the bottle-

		Relative	Utilization (%)				
Scheme	HP:NP	Speedup	Disk	HP	I/O Bus	SW	NP
Base		1.00	32.3	92.9	54.9	15.8	0.1
NITS	1:1	0.90	30.0	76.1	84.6	14.7	11.6
DTS	1:1/9	0.92	29.5	76.5	67.0	14.4	71.4
	1:1/8	0.99	32.0	83.0	72.7	15.7	68.9
	1:1/7	1.07	35.1	90.8	79.6	17.1	66.1
	1:1/6	1.09	35.6	92.2	80.8	17.4	57.6
	1:1/4	1.09	35.6	92.2	80.8	17.4	38.6
	1:1/2	1.09	35.6	92.2	80.8	17.4	20.5
	1:1	1.09	35.6	92.2	80.8	17.4	10.1

Table 6. Comparison of Relative Speed-Up and Resource Utilization: Base, NITS, and DTS schemes

neck and its utilization is lower than that of HP and I/O Bus. When NP's computing power increases from 1:1/6 to 1:1, we see that there are no improvements in the speedup. It can be inferred from the high resource utilization (92.2%) of HP that the tuple processing costs of the host is a dominant factor, which is closely followed by the I/O bus utilization (80.8%). Although we notice a high I/O bus utilization (80.8%) in the DTS scheme, it is relatively lower than that in NITS scheme (84.6%), indicating that I/O bus traffic has decreased in DTS scheme.

Since we observed a high I/O utilization for the DTS scheme, we ran experiments with I/O bandwidth doubled. We observed an increase in speedup from 1.09 to 1.14, suggesting that I/O bandwidth still limits the performance gains achieved due to offloading the tuple processing to NP. This led us to next consider modifications to the cluster node architecture whereby the I/O bus requirements could be reduced.

6 Network Interface with attached Disk (NID) Scheme

Here, we consider a cluster node architecture where the disk is directly attached to the network interface instead of to the system bus as in previous schemes and hence the name Network Interface with attached Disk (NID). Attaching the disk to NI enables direct transfer of tuples from the disk to the network interface rather than through the system I/O bus, thus avoiding the I/O bus bottleneck. This architecture of a cluster with NID is shown in Figure 2. Since our objective is to find out the performance benefits of removing the I/O bus bottleneck, we have assumed that NI has sufficient memory (roughly 4MB) to stage the relations from the disk.

In this scheme, NP reads the tuples directly from the attached disk and computes $node_id$s for all tuples. It then communicates the groups of tuples to the appropriate nodes using the $node_id$s. Further, tuples to be processed locally are DMAed to the host memory from the NI SRAM. The host processor receives tuples from the NI (these could be tuples from the same node or tuples arriving through the network) and processes them, as in the Base scheme.

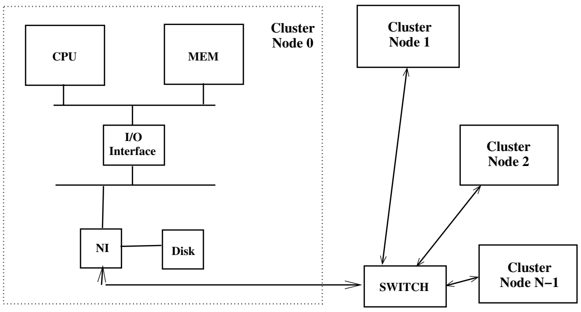

Figure 2. Cluster of Workstations with Disk attached to NI (NID)

		Relative	Utilization (%)				
Scheme	HP:NP	Speedup	Disk	HP	I/O Bus	SW	NP
Base		1.00	32.3	92.9	54.9	15.8	0.1
NID	1:1/9	0.83	27.1	68.8	8.4	13.3	88.3
	1:1/8	0.92	30.0	76.1	9.2	14.7	87.0
	1:1/7	1.03	33.4	84.9	10.3	16.4	84.9
	1:1/6	1.16	37.6	95.5	11.6	18.4	82.1
	1:1/4	1.16	37.7	95.7	11.6	18.4	55.2
	1:1/2	1.16	37.7	95.7	11.6	18.4	28.1
	1:1	1.16	37.6	95.7	11.6	18.4	14.5

Table 7. Comparison of Relative Speed-Up and Resource Utilization: Base and NID schemes

Table 7 shows the relative speedup and resource utilization of NID compared to that of the Base scheme. Attaching the disk to the NI directly results in reduction in I/O Bus utilization from 80.8% to 11.6%. However for *HP:NP* ratios of 1:1/9 and 1:1/8, there is no improvement in execution time. NP is the highest utilized resource and hence offloading tuple processing activities does not reduce the query execution time. Increasing NP computing power to 1/6 of HP increases the speedup of NID scheme to 1.16. At this point HP utilization is high (95.7%) and NP utilization decreases, suggesting that the host processor dominates query exection time. Further increase in NP computing power does not yield additional benefits. This is due to the fact that the tuple processing tasks (related to the join operation) performed by HP dominate the execution time. This motivates us to next look for ways that tuple processing can be offloaded to NP.

7 Network Interface Join (NIJ) Scheme

In this scheme, both HP and NP perform the tuple processing tasks *i.e.,* perform the build and probe operation in each stage. We accomplish this through a simple change in the split hash function. In the proposed scheme, the processor on the network interface (NI) also performs join processing and hence we refer to this scheme as NI Join scheme. This is an extension of the parallel hash join algorithm in which each NP in a *N* node cluster is also a join site, thus the join takes place on *2N* processors. Note that in this scheme the disk is attached to the system bus and HP performs the tuple splitting task[2].

In this scheme HP reads the tuples off the disk and computes the $node\_id$, assuming a *2N* processor system. The tuples are grouped into buckets based on the $node\_id$, and even/odd buckets are distributed to host/NP on each node respectively. When the tuples arrive at HP, they are processed (build/probe) in the usual manner. Those tuples destined for NP are processed just as in HP while the rest are forwarded to the respective destination nodes. Note that since both HP and NP maintain their own hash tables, no synchronization is required between the HP and NP in hash table accesses.

As mentioned earlier, a possible mismatch in the processing powers of HP and NP could lead to performance problems if the load is not balanced between the host and network processors. However this can be controlled by appropriately splitting the number of tuples being processed by the host and NP in the ratio of their respective computing power.

Tables 8 shows the speedup and resource utilizations under the NIJ scheme compared to those of the Base scheme. As the *HP:NP* changes from 1:1/9 to 1:1, the relative speedup of the NIJ scheme improves from 1.05 to 1.47. We make an important observation from Tables 8, viz, at low values of *HP:NP* ($<=$ 1:1/8) ratio, where other scheme show performance degradation (speedup $<$ 1), the NIJ scheme shows a speedup of 1.05 to 1.07. Similarly when *HP:NP* ratio is greater than 1:1/6, the maximum speedup for the best scheme (NID) saturates at 1.16, whereas for the NIJ scheme it increases to 1.47 (for *HP:NP*=1:1), making a strong case for increasing the computing power of NPs.

The resource utilizations reveal interesting aspects of the NIJ scheme. The HP and NP utilization vary from 89.8% to 41.4% and 78.7% to 63.2%, respectively, indicating that HP and NP have nearly balanced work load. In the NIJ scheme, the I/O bus utilization is less than both host and NP utilizations except for *HP:NP* equal to 1 and hence does not limit the performance. Disk and Switch utilizations are also less that 50%. The balanced utilization of resources makes the NIJ scheme attractive from the perspective of achieving scalable performance. Further, if any other query processing, like aggregation or sorting has to executed by HP, more tuples can be diverted to NP so that load balancing can be maintained. Thus, we find NIJ to be the best alternative of all our schemes.

Scheme	HP:NP	Relative	Utilization (%)				
		Speedup	Disk	HP	I/O Bus	SW	NP
Base		1.00	32.3	92.9	54.9	15.8	0.1
NIJ	1:1/9	1.05	34.1	89.8	57.4	16.7	78.7
	1:1/8	1.07	34.4	89.7	57.9	16.8	78.4
	1:1/7	1.08	34.8	89.5	58.5	17.0	78.3
	1:1/6	1.09	35.2	89.3	59.2	17.3	77.7
	1:1/4	1.15	36.9	88.4	61.7	18.1	75.8
	1:1/2	1.28	41.3	85.9	68.4	20.2	70.7
	1:1	1.47	48.5	81.4	79.2	23.7	62.2

Table 8. Comparison of Relative Speed-Up and Resource Utilization: Base and NIJ schemes

8 Scalability of Schemes

The results presented in the earlier sections are from simulations of an 8 node cluster connected by a single switch giving rise to single hop delay. It is also important to study the performance of the proposed schemes with larger cluster sizes, where packets experience multi-hop delays as they traverse multiple switches. In order to account for multi-hop delays, we assume that the nodes are interconnected by a 2D torus network and perform scalability tests for 64 nodes.

We perform two types of simulations on larger cluster sizes: (a) Scaleup test – where the data size is increased as the number of nodes in the cluster is increased, and (b) Speedup test – where the total data size remains constant when the cluster size is increased. We define Scaleup and Speedup for a Scheme A as given by the equations in Figure 3. Note that for the Base scheme a Scaleup of 1 for a N node cluster represents linear speedup.

Table 9 shows the Scaleup and Speedup results for different offloading schemes under various cluster sizes[3]. In the Base scheme as the number of nodes is increased to 64, the Scaleup reaches 0.84. This reduction in Scaleup is due to the overhead in communication. We also see that the Scaleup reaches 1.24 for NIJ scheme for 64 nodes. This is attractive as (i) with the scaled problem size, the NIJ scheme achieves better than linear speedup and, (ii) the Relative speedup for 64 nodes (with respect to the Base Scheme for 64 nodes) is 1.48, similar to what was achieved

[2] It is possible to incorporate the idea of NID in this scheme. We defer this to future work.

[3] We do not report Scaleup and Speedup numbers for DTS, NID, and NIJ schemes for single node case, as tuple splitting is relevant only for multiple nodes

$$Scaleup\ =\ \frac{Execution\ Time\ of\ Original\ Problem\ on\ a\ Single\ node\ for\ Base\ Case}{Execution\ Time\ of\ (linearly)\ Scaled\ Problem\ on\ N\ nodes\ for\ Scheme\ A}$$

$$Speedup\ =\ \frac{Execution\ Time\ of\ Original\ Problem\ on\ a\ Single\ node\ for\ Base\ Case}{Execution\ Time\ of\ Original\ Problem\ on\ N\ nodes\ for\ Scheme\ A}$$

Figure 3. Definition for Scaleup and Speedup for a Scheme A

		Scaleup						Speedup					
Scheme	HP:NP	1	2	4	8	16	64	1	2	4	8	16	64
Base		1.00	0.92	0.88	0.86	0.85	0.84	1.00	1.83	3.51	6.89	13.58	54.33
DTS	1:1/6	-	0.96	0.94	0.93	0.93	0.93	-	1.93	3.78	7.48	14.97	58.68
	1:1	-	0.96	0.94	0.93	0.93	0.93	-	1.93	3.78	7.48	14.97	58.68
NID	1:1/6	-	1.01	1.00	1.00	0.99	0.99	-	2.02	4.02	8.02	15.95	63.78
	1:1	-	1.01	1.00	1.00	0.99	0.99	-	2.02	4.02	8.02	15.95	63.78
NIJ	1:1/6	-	0.99	0.96	0.94	0.93	0.92	-	1.99	3.84	7.52	14.97	58.68
	1:1	-	1.32	1.28	1.26	1.25	1.24	-	2.65	5.15	10.12	20.10	81.50

Table 9. Scalability: Relative Speedup of Base, DTS, NID, and NIJ schemes

for 8 nodes (refer to Table 8). In the Speedup experiment, we achieve a speedup of 54.33 for 64 nodes for the Base scheme. This corresponds to an efficiency (Speedup/N) of 0.85. Again for DTS, NID and NIJ schemes the speedup increases to 58.68, 63.78, 81.50 respectively and the corresponding relative speedup metric is 1.08, 1.17, 1.50 respectively. These values which are for 64 nodes are similar to those for 8 nodes (refer to Tables 6, 7 and 8).

We found that NIJ exhibits balanced utilizations of resources as compared to the other schemes for larger clusters also, which makes it a good choice for larger clusters. Further, we note that switch utilization is less than 25%, implying that interconnection network is not a bottleneck resource. Thus, our Scaleup and Speedup tests indicate that the proposed offloading schemes give good performance benefits for clusters of larger sizes.

9 Related Work

In the database domain, various architectures have been based on the idea of off-loading processing from the host processor [2, 15, 11]. Active Disks assume that each disk drive has reasonable processing power and memory, so that application-specific code can be downloaded and executed on data off the disk [2]. The application is split into a disk-resident and host-resident code which communicate using streams. Riedel *et al* report performance from running database, data mining and multimedia applications on Active Disk architecture [15]. Memik *et al* evaluate a smart disk cluster, where each disk has an embedded processor, controller, disk and memory [11]. In order to avoid unwanted data transfer from disk, the host processor analyzes the query for bindable operations and generates a new bundled query execution plan. The host processor then passes control messages to the smart disk indicating the order of executing operations in the bundle. These approaches are disk centric, while ours is NIC centric – they off-load higher level database operations to the processor on the disk drive, while we off-load operations that are communication related to NIC.

Krishnamurthy *et al* developed a network interface (NI) architecture where the co-processor controls both communication links and SCSI disk interface [9]. They use this architecture to build scalable media severs which stream media frames at real time rates to clients. The heart of this system is a media stream scheduler implemented on the co-processor which can stream data from the disks attached to the NI directly to the clients. This reduces the host bus memory traffic and decreases the host processor utilization. Our NID model, is similar; tuple processing activities are offloaded to the network processor. This also prevents the host I/O bus from being a bottleneck. In the work by Binkert *et al* [3], the communication overhead is found to be high due to the standard I/O bus. They propose alternatives of how NIC can be closely coupled to CPU to reduce the driver overhead through a detailed performance evaluation. Our NID scheme is similar in spirit in that we attach the disk to the NI so that I/O bus bottleneck is be avoided.

10 Conclusions

Today, workstation clusters are a cost effective solution for large database query processing applications. Optimizing the performance of these parallel query processing clusters is commercially important. We propose both software and hardware modifications exploiting the programmable features of NP to achieve higher performance with balanced utilization of system resources. Maintaining balanced system utilization is important for scalable growth, but not easy to ensure. We evaluate the performance of the proposed modifications using a validated timed Petri net models. We find that certain modifications, like offloading tuple splitting work from the host processor to the network processor, results in execution time speedup of upto 1.16. By offloading a part of join processing to NP, our NI Join (NIJ) scheme not only improves the execution time significantly over the Base scheme, but also achieves balanced system utilization. We suggest that if future network interfaces are equipped with programmable processors of higher processing power, applications should be able to exploit them in improving system performance. We find that the proposed schemes demonstrate near linear speedups for the scalability tests which ensures their usefulness for cluster of large sizes.

As mentioned earlier, we have assumed that data is distributed uniformly on the join attribute. As future work, we plan to study the effect of skew in data distribution on the performance of these schemes. Another interesting aspect is to study how the schemes we proposed perform under different query execution plans such as right-deep or bushy-trees.

Acknowledgments

This work was partially supported by a research grant from IBM India Research Center. We would like to thank Prof. W. M. Zuberek for the CNET simulation software.

References

[1] T. Anderson, D. Culler, and D. Patterson. A Case for NOW (Networks of Workstations). *IEEE Micro,* 16(1):54-64, Feb 1995.

[2] A. Acharya, M. Uysal, and J. Saltz. Active Disks: Programming Model, Algorithms and Evaluation. In *Proc. of 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 81-91, Oct 1998.

[3] N. L. Binkert, L. R. Hsu, A. G. Saidi, R. G. Dreslinski, A. L. Schultz, and S. K. Reinhardt. Analyzing NIC Overheads in Network-Intensive Workloads. In *8th Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW)*, Feb 2005

[4] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using Segmented Right-Deep Trees for the Execution of Pipelined Hash Joins. In *Proc. of 18th Very Large Data Bases*, pages 15-26, Aug 1992.

[5] D. DeWitt and J. Gray. Parallel Database Systems: The future of High Performance Database Systems. *Communications of the ACM* 35(6):85-98, Jun 1992.

[6] D. DeWitt, R. Katz, F.Olken, L.Shapiro, M.Stonebraker and D.Wood. Implementation Techniques for Main Memory Database System. In *Proc. of ACM SIGMOD Conference*, Jun 1984.

[7] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User Level Interwork Interface for Parallel and Distributed Computing. In *Proc. of 15th ACM Symp. on Operating Systems Principles*, Dec 1995

[8] R. Govindarajan, F. Suciu, W. M. Zuberek. Timed Petri Net Models of Multithreaded Multiprocessor Architectures. In *Proc. of 9th Intl. Workshop on Petri Nets and Performance Models*, Jun 1997

[9] R. Krishnamurthy, K. Schwan, R. West, M. C. Rosu. On Network CoProcessors for Scalable, Predictable Media Services.*IEEE Trans. on Parallel and Distributed Systems*, 14(7):655-670, Jul 2003.

[10] J. Liu, B. Chandrasekaran, W Yu *et al* . Microbenchmark Performance Comparison of High-Speed Cluster Interconnects. *IEEE Micro,* 24(1):42-51, Jan-Feb 2004.

[11] G. Memik, M. T. Kandemir and A. Choudhary. Design and Evaluation of Smart Disk Cluster for DSS Commercial Workloads. *Journal of Parallel and Distributed Computing (JPDC)*, 62(11):1633-1664, 2001.

[12] P. Mishra and M. H. Eich. Join Processing in relational databases. *ACM Computing Surveys*, 24(1):63-113, Mar 1992.

[13] Myricom Inc., LANai 7, Myricom Draft, Jun 1999.

[14] PCI-SIG Home, http://www.pcisig.com/, 2004.

[15] E. Riedel, G. Gibson, and C. Faloutsos. Active storage for large-scale datamining and multimedia. In *Proc. of 24th Intl. Conf. on Very Large Data Bases*, 62-73, Aug 1998.

[16] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. of ACM SIGMOD Conference*, Jun 1989.

[17] D. Schneider and D. DeWitt. Tradeoffs in processing complex queries via hashing in multiprocessor database machines. In *Proc. of 16th Intl. Conf. on Very Large Data Bases*, Aug 1990.

[18] C. L. Seitz. Myrinet Technology Roadmap. *Myrinet Users Group Conf.* May 2002.

[19] V. Karamcheti, and A. Chien. Software Overhead in Messaging Layers: Where Does the Time Go? In *Proc. of 6th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct 1994.

[20] T. Tamura, M. Oguchi, and M. Kitsuregawa. Parallel Database Processing on a 100 Node PC Cluster: Cases for Decision Support Query Processing and Data Mining. In *Proc. of Supercomputing*, Nov 1997.

[21] TPC Benchmark^{TM} H (Decision Support) Standard Specification Revision 1.3.0. Transaction Processing Performance Council(TPC), 1999.

[22] W. M. Zuberek. Modeling using Timed Petri Nets - event-driven simulation, Technical Report No. 9602, Dept. of Computer Science, Memorial Univ. of Newfoundland, St. John's, Canada, 1996 (`ftp://ftp.ca.mun.ca/pub/techreports/tr-9602.ps.Z`).