A Framework for Collective Personalized Communication

Laxmikant V. Kalé, Sameer Kumar
Department of Computer Science
University of Illinois at Urbana-Champaign
{kale,skumar2}@cs.uiuc.edu

Krishnan Varadarajan
Microsoft Inc.
krishvar@windows.microsoft.com

Abstract

This paper explores collective personalized communication. For example, in all-to-all personalized communication *(AAPC), each processor sends a distinct message to every other processor. However, for many applications, the collective communication pattern is many-to-many, where each processor sends a distinct message to a subset of processors. In this paper we first present strategies that reduce per-message cost to optimize AAPC. We then present performance results of these strategies in both all-to-all and many-to-many scenarios. These strategies are implemented in a flexible, asynchronous library with a non-blocking interface, and a message-driven runtime system. This allows the collective communication to run concurrently with the application, if desired. As a result the computational overhead of the communication is substantially reduced, at least on machines such as PSC Lemieux, which sport a co-processor capable of remote DMA. We demonstrate the advantages of our framework with performance results on several benchmarks and applications,*

1 Introduction

The communication cost of a parallel application can greatly affect its scalability. Although communication bandwidth increases have kept pace with increases in processor speed over the past decade, the communication latencies (including the software overhead) for each message have not decreased proportionately. The past decade has also seen the increase in popularity of workstation clusters. Such systems can be cost effective but use general purpose operating systems and tend to have higher communication latencies and per message overheads.

Collective communication operations, which often involve most processors in a system, are time consuming and can involve massive data movement. An inefficient implementation of such operations will often affect the performance of the application significantly. In this paper we focus on *collective personalized communications*, where each processor sends distinct data to many other processors.

An *all-to-all* personalized communication (AAPC for brevity) is a collective operation in which *each* processor sends distinct data to *all* other processors. AAPC has been extensively studied in literature. Many of these optimization algorithms have been designed for particular network topologies, including mesh [1, 7, 19, 17, 22], torus [10, 18, 4], fat tree [16, 3] etc. AAPC algorithms can be classified as direct or indirect[20]. With direct algorithms each processor sends its data directly to the destination processors. Direct algorithms aim at exploiting specific communication architectures and emphasize sequencing of messages to avoid link and node contention. In the indirect approach [2] processors combine messages into larger data blocks which are sent to intermediate processors to be routed to the final destination. Indirect approaches allow message combining, and are are hence most useful for small messages; while direct approaches minimize the number of copies and so are suitable for long messages [1, 20].

Many applications require a *many-to-many* personalized communication (MMPC for brevity), where many (not all) processors send personalized data to many (not all) other processors. For example the molecular dynamics application Namd [15] has a transpose-like operation in which 192 (out of possibly a few thousand) processors send messages to 144 processors. With MMPC the best strategy not only depends on the size of message but also on the degree of the communication graph (δ), which is the number of processors each processor sends its data to. In this paper we study how existing strategies for AAPC perform with MMPC.

We limit our study here to indirect strategies. A basic issue in collective communication with small messages is the per-message cost (α). The strategies we describe aim at reducing the α cost by sending fewer, larger messages. A virtual topology is imposed over the processors. Messages destined to a group of processors are combined into one message and sent to a representative processor in the group. The representative, acting as an intermediary, combines messages going to the same destination, and forwards

them. We can even repeat this combining-and-forwarding "phase" several times. We will describe three such multi-phase strategies for personalized collective communication, (i) Mesh, (ii) 3d Grid, (iii) Hypercube.

Our framework library is implemented on top of Converse[8] which is portable and available on most platforms. Charm++ [9] and MPI programs can use our library through interfaces to Converse. A synchronous, blocking implementation like MPI_Alltoall is potentially inefficient because processors will idle unnecessarily, until the communication is complete (from the local point of view). Moreover all the processors may not be involved in every stage of the strategy. In the above Namd example some processors (around 48) may be idle during the different stages of the communication operation. Our library provides an asynchronous, non-blocking, interface. This gives the application the flexibility to use the idle CPU time for other useful computation. As a result, it can effectively handle the scenario when only a subset of processors are involved in the collective communication operation.

MPI applications can take advantage of our library via a split-phase interface. This enables programs to start a collective operation and then poll the system till it finishes. In the meantime the program can do computation.

The rest of the paper is organized as follows : Section 2 describes the various strategies used by the library. Section 3 discusses the MMPC problem and how the strategies developed for AAPC perform with MMPC. Section 4 provides performance results of the library with several benchmark applications. Related work is presented in Section 5, while Section 6 presents some concluding remarks.

2 All to All Personalized Communication

This section describes strategies to optimize AAPC. We first present a simple communication model used for analyzing our strategies. We then present three indirect strategies which use message combining for optimizing AAPC.

2.1 Communication Model

Many models for communication on parallel architectures have been presented [13, 21, 2, 20]. We use a simplified communication model [2, 20] where the time to send a point to point message is given by

$$T_{ptp} = \alpha + m\beta$$

where α is the sending, receiving and network overhead for the message, β is the per unit data transfer cost and m the size of the message.

However, we note that in presence of a communication co-processor, it is desirable to distinguish the cost to the processor from the latency of the operation.

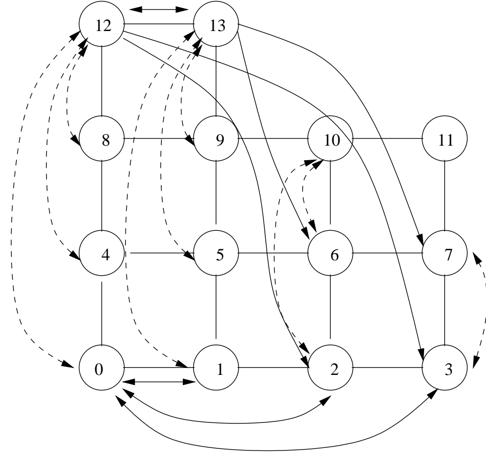

Figure 1. Mesh based AAPC, with holes.

$$T_{ptp_{cpu}} = \alpha_{cpu} + m\beta_{cpu}$$

On machines with a co-processor capable of remote DMA, β_{cpu}, the per-byte cost incurred by the processor is substantially lower.

With a direct implementation of AAPC, using individual point-to-point messages, each processor has a cost given by

$$Cost = (P - 1) \times (\alpha + m\beta) \qquad (1)$$

Here P is the total number of processors. Network contention is not modeled here because for small messages the cost is dominated by the software overhead or the α cost. Thus, for $\alpha = 5\mu s, \beta = 3.33 ns/byte$ and $m = 100 bytes$, on 1000 processors this operation would take 5.33 ms [1]. Strategies based on considerations of physical topology are also out of scope of this paper.

In the indirect strategies we discuss next, each processor combines messages destined to a group of processors into one message, and sends a combined message to one (or a series of) intermediary(ies). The strategies thus aim at reducing the alpha component of the communication cost while typically trading off an increase in the β component.

Next, we will consider three virtual topologies: 2-D mesh, 3-D grid and hypercube. We will refer to *2-D mesh* by mesh and *3-D grid* by grid in the remainder of the paper.

2.2 2-D Mesh

In this scheme, the messages are routed along a 2-D mesh. In the first phase of the algorithm, each node sends

[1] The *all-to-all* time on 1024 processors presented in Section 4 is more than this in part because we are using 4 processors per node on Lemieux and messages from processors will have to wait for messages from the other processors in that node.

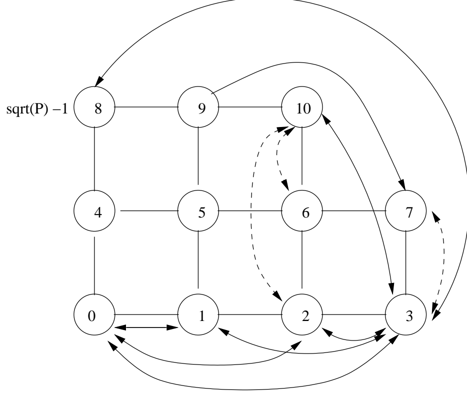

Figure 2. **Mesh based AAPC: Worst case with holes**

the messages to all the nodes in its row. In the second phase, the nodes sort these messages and send them to the nodes in their column. Thus all the messages travel at most two hops before reaching the destination. Here each processor sends $\sqrt{P} - 1$ messages of size $\sqrt{P} \times m$ bytes in each of the two phases. The completion time for AAPC with mesh strategy, T_{mesh} is shown in equation 2.

$$
\begin{aligned}
T_{mesh} &= 2 \times (\sqrt{P} - 1) \times (\alpha + \sqrt{P} m\beta) \\
T_{mesh} &\approx 2\sqrt{P}\alpha + 2Pm\beta
\end{aligned} \tag{2}
$$

When the number of nodes is not a perfect square, the mesh is constructed using the next higher perfect square. This gives rise to *holes* in the mesh. Figure 1 illustrates our scheme for handling holes, in a mesh with two holes. (The dotted arrows show the second stage.) The role assigned to each hole (which are always in the top row) is mapped uniformly to the remaining processors in its column. So if a node (i, j) needs to send a message to column k and node (i, k) is a hole, it sends the message to node $(j\%(NROWS - 1), k)$. Here NROWS is the number of rows in the mesh. Thus in the first round node 12 sends messages to nodes 2 and 3. No messages are sent to a row with no processors in it. Dummy messages are used in case a processor has no data to send.

Notice that $\lceil\sqrt{P}\rceil - 1 \leq NROWS \leq \lceil\sqrt{P}\rceil$, whereas number of columns is always $\lceil\sqrt{P}\rceil$. (If $NROWS \leq \lceil\sqrt{P}\rceil - 1$ then the next smaller square mesh would have been used). Thus the number of processors that would have sent messages to the hole is at most $\lceil\sqrt{P}\rceil - 1$, and the processors in the hole's column (that share its role) is at least $(NROWS - 1) = \lceil\sqrt{P}\rceil - 2$. Hence the presence of holes will increase the number of messages received by processors in columns containing holes by one (nodes 2,3,6,7 in figure 1) or two (node 3 in figure 2). Figure 2 shows the worst case scenario when a processor (3) receives two extra messages. The worst case happens when the number of rows is $\lceil\sqrt{P}\rceil - 1$ and there is only one hole.

In the second phase (when processors exchange messages along their columns) these processors will exchange one or two messages less and the total will remain unchanged. So the α factor of equation 2 remains the same and the β factor will only increase by $2(\sqrt{P}).m.\beta$ which can be ignored for large P. Thus leaving equation 2 unaltered. A simple proof can show that the cost of equation 2 is within the optimal cost for any mesh by a constant additive factor in the number of messages.

2.3 3D Grid

We also implemented a virtual grid topology. In this topology messages are sent in three phases along the X, Y and Z dimensions respectively.

In the first phase, each processor sends a message to its $\sqrt[3]{(P)} - 1$ neighbors along the X dimension. The data sent contains the messages for the processors in the plane indexed by the X coordinate of the destination. In the second phase, messages are sent along the Y dimension. The messages contains data for all the processors that have the same X and Y axes but different Z axis as the destination processor. In the third and final phase data is communicated to all the Z axis neighbors. Since in each phase the processor sends at most $\sqrt[3]{P}$ messages the total time for each operation is given by equation 3.

$$
\begin{aligned}
T_{grid} &= 3 \times (\sqrt[3]{P} - 1) \times (\alpha + (\sqrt[3]{P})^2 m\beta) \\
T_{grid} &\approx 3\sqrt[3]{P}\alpha + 3Pm\beta
\end{aligned} \tag{3}
$$

When the number of processors is not a perfect cube, the next larger perfect cube is chosen. Here we use a simpler strategy to map holes for ease of implementation. Holes are mapped to the corresponding processor in the penultimate plane. So all messages destined to a hole are received by the corresponding processor in the penultimate plane. If there is only one plane the holes are mapped like they are in the mesh strategy. No messages are sent to planes with no processors in them. Here each processor can receive messages for a hole in the first and second phases. We also assume that the receiving α cost is half the total α cost. The cost for grid strategy with holes is given by equation 4. Here λ_h is 1 if there are holes in the grid, 0 otherwise.

$$
T_{grid} \approx (3 + \lambda_h) \times \sqrt[3]{P}\alpha + (3 + 2\lambda_h) \times Pm\beta \tag{4}
$$

2.4 Hypercube

The hypercube (Dimensional Exchange) scheme consists of $log_2(P)$ stages. In each stage, the neighboring nodes in one dimension exchange messages. In the next stage, these messages are combined and exchanged between the nodes in the next dimension. This continues until all the dimensions are exhausted.

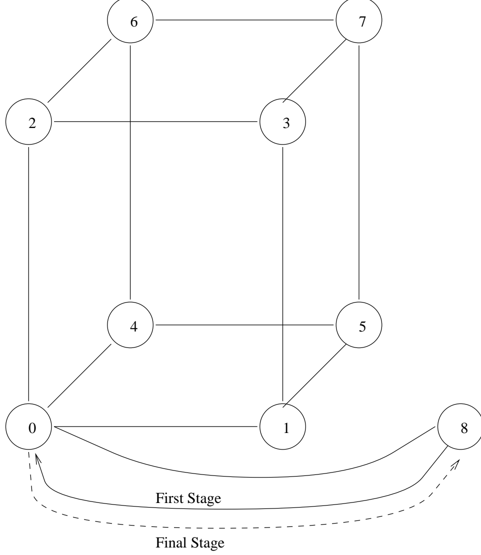

Figure 3. Hypercube based AAPC with an imperfect hypercube.

During each of the $log_2(P)$ phases, the processors along one dimension exchange data. Thus in phase 1, each processor combines the messages for P/2 processors and sends it to its neighbor in that dimension. In the second phase, the messages destined for $P/4$ processors are combined. But now, each processor has the data it received in phase 1 in addition to its own data. Thus it combines $2 \times (P/4) \times m$ bytes and sends it to its neighbor. The overall cost is represented by the equation 5.

$$T_{hypercube} = log_2 P \times (\alpha + \frac{P}{2} m\beta) \tag{5}$$

In the case of an imperfect hypercube (when the number of nodes is not a power of 2), the next lower hypercube is formed. In the first step, the nodes that are outside this hypercube send all their messages to their corresponding neighbor in this hypercube. For example, in Figure 3, node 8 sends it messages to node 0 in the first stage. As in the usual scheme, dimensional exchange of messages happens in this hypercube. All the messages for node 8 are sent to node 0. In the final stage, node 0 combines all the messages for node 8 and sends it to node 8. If there are few holes many processors will have twice the data to send. The cost of hypercube with holes is shown in equation 6

$$T_{hypercube} \approx log_2 P \times (\alpha + (\frac{1 + \lambda_h}{2}) \times Pm\beta) \tag{6}$$

Mesh tends to be faster then hypercube because it has fewer stages and will load the network less. Hypercube will transmit $log_2 P$ times the total data exchanged on to the network. Grid will have an intermediate performance.

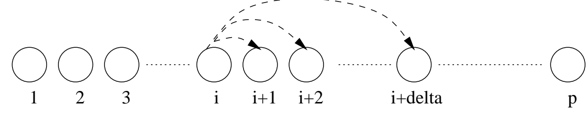

Figure 4. Neighbor Send Application

3 Many to Many Personalized

For many applications each processor in the collective operation may not send messages to *all* other processors. Applications that have such a communication pattern include Namd [15], Barnes Hut Particle simulator[23], Euler Solver and Conjugate grid solver. Hence the degree (δ) of the communication graph, which is the number of remote processors each processor communicates with, becomes another important factor in the cost equations. We comment below on two classes of the MMPC (*many-to-many personalized communication*) pattern.

3.1 Uniform Many-to-Many Personalized

In uniform MMPC, each processor sends (and receives) around the same number of messages. The AAPC applications are a special case of this class. Many other applications which have small variances in the degree (δ) also belong to this class. An example of uniform MMPC is the neighbor-send application shown in figure 4. Here processor i sends messages to processors $(i + k)\%P$ for $k = 1, 2, .., \delta$.

For MMPC, the cost equations of section 2 are modified:

$$\begin{aligned} T_{mesh} &\approx 2\sqrt{P}\alpha + 2\delta m\beta && (7) \\ T_{grid} &\approx 3\sqrt[3]{P}\alpha + 3\delta m\beta && (8) \\ T_{hypercube} &\approx log_2 P \times (\alpha + \delta m\beta) && (9) \\ T_{direct} &= \delta \times (\alpha + m\beta) && (10) \end{aligned}$$

In the mesh strategy, each processor sends δ messages and each of these messages is transmitted twice on the network. So the amount of per-byte cost spent on all the messages in the system is $2P\delta m\beta$. Since the MMPC is uniform this cost can be evenly divided among all the processors. The resulting cost equation is given by 7. By a similar argument we get equations 8, 9. Also notice that δ appears only in the β part of the equations 7,8,9. This is because in each virtual topology the number of messages exchanged between the nodes is fixed. If there is no data to send, at least dummy messages need to be sent.

3.2 Non-Uniform Many-to-Many Personalized

In non-uniform MMPC there is a large variance in the number of messages each processor sends (or receives). (E.g. some processors may be the preferred destinations of all the messages sent). There may also be a variance in the *sizes* of messages processors exchange. Hence a more general framework needs to be developed which should also consider actual destinations of the message along with the degree of the graph. If certain nodes in the virtual topology are being overloaded then the virtual topology may have to be rearranged. Large messages can be handled by sending them directly and not using an intermediate processors to route it, thus saving on the β cost. We are still investigating this problem. However, the 3-D FFT application described in 4.4 explores performance of strategies for this pattern.

4 Performance

We tested the performance of our library on PSC Lemieux[11], a 750 node, 3000 processor cluster. Each node in Lemieux is a Quad 1Ghz Alpha server connected by Quadrics Elan [14], a high speed interconnect with a $4.5\mu s$ message latency.

Figures 6 and 7 present the performance of AAPC using our library and MPI on Lemieux, using 4 processors per node. Mesh and 3d Grid do better than direct sends for messages smaller than 1000 bytes on both 512 and 1024 processor runs. For very small messages these indirect strategies are better than MPI all-to-all. For intermediate message sizes however MPI is somewhat better.

Also notice the jump for direct sends at message size of 2KB. This is because our runtime system switches from statically to dynamically allocated buffers at this point. MPI has a similar and much larger jump, which further increases with the number of processors.

Although the indirect strategies are clearly better than direct sends for messages smaller than 1KB, for a small range of message sizes MPI does better than our strategies. However, two factors make our library superior to MPI.

Scalability: Figure 5 shows the *scalability* of our library compared with MPI for the all to all operation with a message size of 76 bytes. The Hypercube strategy does best for a small number of processors (this is not clearly seen in the linear-linear graph). But as the number of processors increase, mesh and 3d grid improve, because they use only two and three times the total amount of network bandwidth respectively, while for hypercube the duplication factor is $\log p/2$. MPI compares well for a small number of processors but for 64 or more processors our strategies start doing substantially better (e.g. 55 ms vs 32 ms on 2k processors).

CPU Overhead: Probably the most significant advantage of our strategy arises from its use of a message-driven substrate on machines with a communication co-processor. In contrast to MPI, our library is asynchronous (with a non-blocking interface), and allows other computations to proceed while AAPC is in progress. Figure 8 displays the amount of *CPU time* spent in the AAPC operations on 1024 processors. This shows the software overhead of the operation incurred by the CPU. Note that this overhead is substantially less than the overall time for our library. E.g. at 8KB, although the mesh algorithm takes about 800 ms to complete the AAPC operation, it takes less than 32 ms of CPU time away from other useful computation. This is possible because of the remote DMA engines provided by Quadrics Elan cards. The fact that communication co-processors are exploited better by a message-driven system was pointed out by our earlier simulation work ([6, 5].)

In our implementation, we have two calls for the AAPC interface. The first one schedules the messages and the second one polls for completion of the operation. On machines with support for "immediate messages" — those that will be processed even if normal computation is going on — and on message-driven programming models (such as Charm++), this naturally allows for other computations to be carried out concurrently. In other contexts, user programs or libraries need to periodically call a polling function to allow the library to process its messages.

Another interesting perspective is provided by the performance data on on 513 processors with 3 processors per node, shown in Figure 9. Note that all the strategies perform much better here (compare with Figure 7). We believe this is due to OS and communication library interactions when all 4 processors on a node are used (as bourne out by our recent scalability studies [15]).

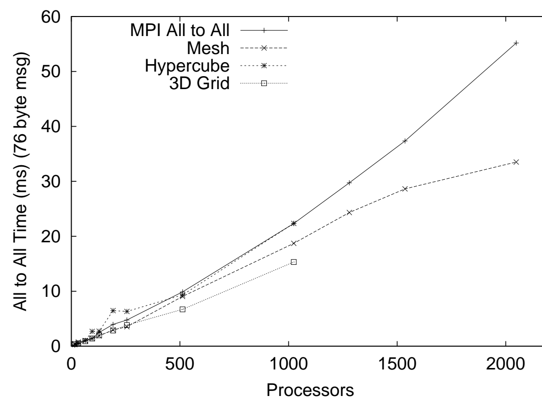

Figure 5. AAPC time for 76 byte message

We also tested our communication library with four benchmark applications, described below.

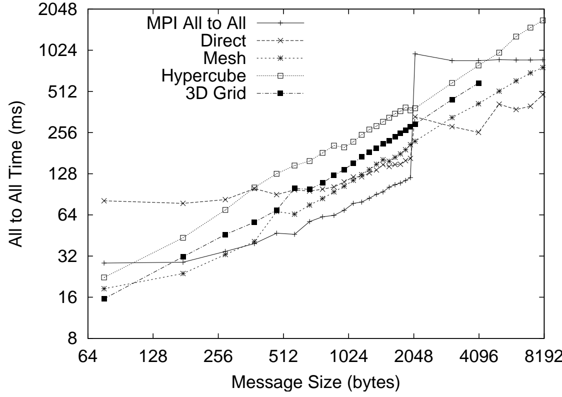

Figure 6. AAPC completion time on 1024 processors

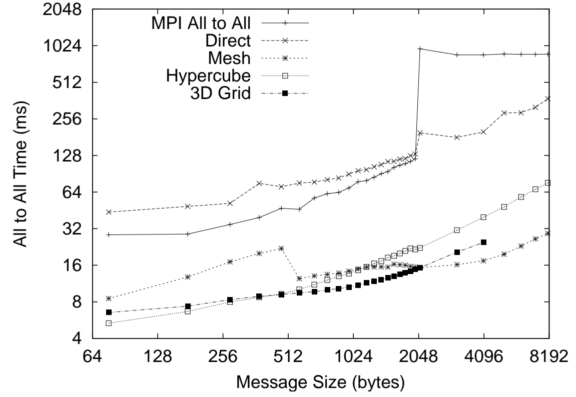

Figure 8. AAPC CPU time on 1024 processors

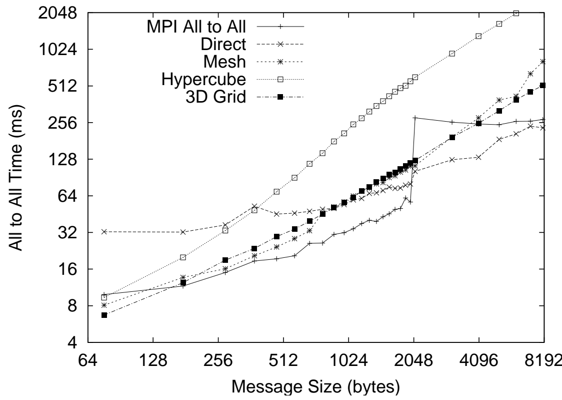

Figure 7. AAPC completion time on 512 processors

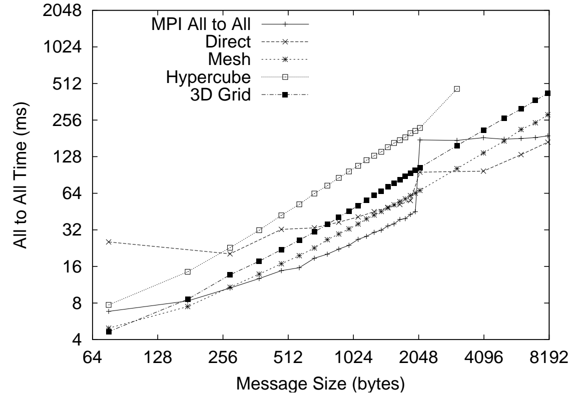

Figure 9. AAPC completion time on 513 processors

4.1 Neighbors

In this benchmark, each processor i sends δ messages to processors $(i+k)\%P$ for $k = 1, 2, .., \delta$. Figures 10 and 11 show the performance of the strategies with the degree of the graph δ being varied from 64 to 2048, for 2048 processors. For small δ the direct strategy is the best. Comparison between the mesh and grid strategies is interesting. The former sends each byte twice, while the latter sends each byte thrice. But the α cost encountered is lesser when the grid strategy is used. For small(76 byte) messages, the α (per-message) cost dominates and the grid strategy performs better. For larger(476 byte) messages, the grid strategy is better until the degree is 512. For larger degrees, the increased amount of communication volume leads to dominance of the β (per-byte) component, and so the mesh strategy performs better.

4.2 Radix Sort

Radix sort is a sorting program which sorts a random set of 64 bit integers, which is useful in operations such as load balancing using space-filling curves. The initial list is generated by a uniform random number generator. The program goes through four similar stages. In each stage the processors divide the local data among 65536 buckets based on the appropriate set of 16 bits in the 64 bit integers. The total bucket count is globally computed through a reduction and each processor is assigned a set of buckets, in a bucket map which is broadcast to all the processors. All the processors then send the data to their destination processors based on the bucket map. This permutation step involves an AAPC and has the most communication complexity. Radix sort is therefore a classic example of AAPC. The performance of Radix sort with the mesh and direct strategies on 128, 256 and 1024 processors is shown in Table 1. N is the number of integers per processor. Notice that the performance gain of the mesh-based version increases with the

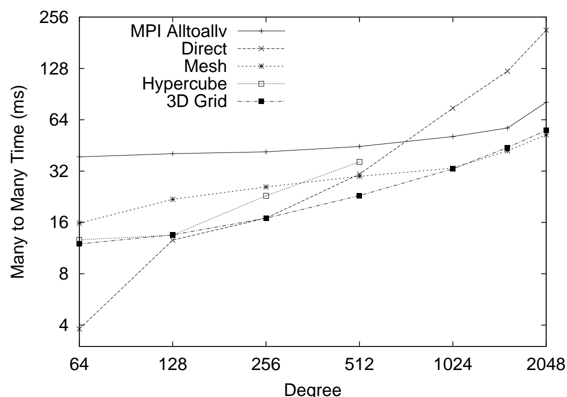

Figure 10. MMPC: time with 76 byte message on 2048 processors

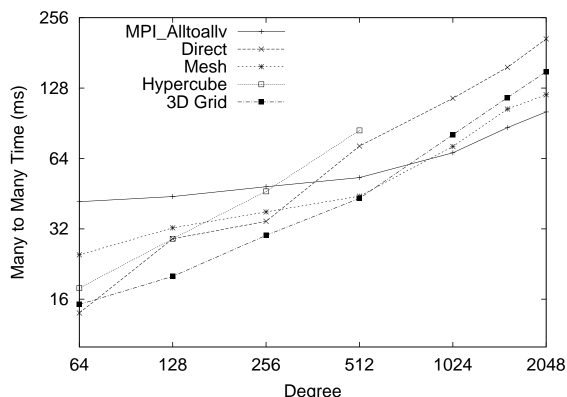

Figure 11. MMPC: time with 476 byte message on 2048 processors

number of processors.

4.3 Namd

Namd is a parallel, object-oriented molecular dynamics program designed for high performance simulation of large bio-molecular systems [15]. Namd employs the prioritized message-driven execution capabilities of the Charm++/Converse parallel runtime system, allowing excellent parallel scaling on both massively parallel supercomputers and commodity workstation clusters.

PME (Particle Mesh Ewald) is one of the important operations in Namd. It involves a 3d FFT. In PME a local 2d FFT is performed by each processor on the Y and Z dimensions of the grid, and the grid is then redistributed along the Y axis for a final 1d FFT on the X dimension. The transformed grid is then multiplied by the appropriate Ewald electrostatic kernel, and a backward FFT performed on the first dimension. The grid is redistributed back along its first dimension, and a backward 2d FFT performed, producing real-space potentials. The structure of PME calculation in

N	Mesh(s)			Direct(s)		
	128	256	1024	128	256	1024
1k	0.57	0.815	1.224	1.39	2.195	4.64
10k	0.61	0.821	1.631	1.394	2.582	9.94
100k	1.21	1.487	2.102	1.455	2.493	11.26
500k	3.39	3.51	4.372	3.561	4.22	16.21
1m	6.45	6.512	7.49	6.714	7.698	18.72
2m	13.6	15.30	14.57	12.25	14.039	30.87

Table 1. Sort Completion Time (s)

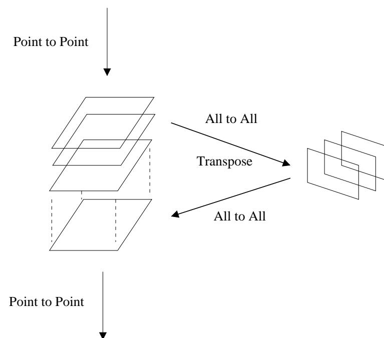

Figure 12. PME calculation in Namd

Namd is shown in figure 12.

Namd simulations were done for two molecules ApoA-1 and ATPase. For ApoA-1 a $108 \times 108 \times 80$ grid is used and for ATPase it was a $192 \times 144 \times 144$ grid. The PME calculation involved a collective communication between the X planes (planes with the same X coordinate) and the Y planes (planes with the same Y coord.). In our large processor runs the number of processors for PME is $max(\#XPlanes, \#YPlanes)$, which is 108 for ApoA-1 and 192 for ATPase. The size of the messages in the collective communication operation is around 900b. Namd was recompiled to use the mesh strategy. The resultant performance gains are presented in Table2. Namd carries out other force computations (for atoms within a cutoff radius) concurrently with PME, and thus is able to take advantage of the lower computation overhead (Fig. 8) of our strategies.

Processors	ApoA-1(ms)		ATPase(ms)		
	Mesh	Direct	Mesh	Direct	MPI
256	39.23	44.40	113.58	120.84	134.53
512	23.37	27.95	60.75	62.96	69.50
1024	20.27	26.754	35.84	38.62	39.31

Table 2. Namd step time (ms)

4.4 3-D FFT

The 3d FFT benchmark performs a similar FFT computation as Namd. First a 2d FFT is performed on each YZ plane and then each Z pencil (a column along the Z direction) in the YZ plane is sent out for a 1d FFT along the X direction. After the 3d FFT step each processor signals completion through a barrier reduction and then the process is repeated again. The performance of 3d FFT is shown in tables 3 and 4.

In tables 3 and 4 the size of the X dimension is 256 and corresponds to the number of senders of the transpose data. The size of the Y dimension corresponds to the number of messages being sent by each processor and the number of receivers of the transpose data. The size of the Z dimension corresponds to the size of the message being sent. The variance in the degree of the communications graph is high if the number of senders is much larger than the number of receivers. In this case all processors will send #Y number of messages to #Y receivers which will receive P messages.

It can be noticed that for the high variance case hypercube does better than expected compared to mesh and 3d grid. We believe this is because in the hypercube strategy each message does not travel $log(P)$ hops and so its β cost is a weak upper-bound. With low variance (which also has a high δ) the behavior is more like uniform many-to-many communication with $mesh < 3dgrid < hypercube$ for most cases as expected. Also, in almost all cases, at least one indirect strategy is better than direct sends.

#Y	#Z	Size	Dir.	Mesh	Hyp.	Grid
256	32	640b	108.9	32.1	42.0	33.6
224	32	640b	78.1	37.0	40.0	29.0
192	32	640b	52.7	34.4	36.1	39.7
256	64	1280b	112.8	40.1	86.7	82.7
224	64	1280b	90.9	41.9	75.6	42.3
192	64	1280b	64.0	44.1	63.4	40.0
256	256	4KB	144.2	141.7	211.4	183.6

Table 3. 3d FFT step time (256 processors and low variance) (ms)

#Y	#Z	Size	Dir.	Mesh	Hyp.	Grid
32	32	640b	20.51	44.25	23.4	35.1
64	32	640b	32.19	51.49	14.4	57.8
128	32	640b	32.12	62.24	20.0	35.7
32	64	1280b	17.65	48.95	19.7	66.3
64	64	1280b	31.67	48.17	20.6	42.1
128	64	1280b	30.21	37.09	30.4	45.2

Table 4. 3d FFT step time (256 processors and high variance) (ms)

5 Related Work

All to all personalized communication has been studied extensively over the past decade. Both direct and indirect optimizations for specific architectures like 2d meshes, tori, 3d grids, hypercubes and fat trees have been presented in [20, 10, 18, 1, 16, 7, 4, 3, 19, 17]. A survey of communication algorithms for AAPC algorithms is presented in [12]. [2] talks about an architecture independent message combining for all to all personalized communication. The paper describes the ring, mesh and 3d grid strategies. Finding dimensions of virtual topologies for non powers of two number processors is not clearly mentioned. Handling non uniform collective communication communication is mentioned in [16]. A hybrid algorithm that combines a direct and an indirect strategies is presented in [19]. It combines the direct Scott's [17] optimal 2d mesh communication strategy with the recursive partitioning strategy which is similar to our hypercube. The effectiveness of pipelining and packetization in direct strategies is presented in [20].

Our work differs from the above because we handle the more general problem of many to many personalized communication where each processor sends messages to a variable number of other processors. Our mesh strategy performs close to the optimal mesh allocation. Our framework is also asynchronous and nonblocking giving more flexibility to the application.

6 Summary and Future Work

In this paper we described three different collective communication algorithms: mesh, 3d grid and hypercube. Our implementations of these algorithms place no restrictions on the number of processors. Our mesh algorithm is within the optimal by an additive constant. We also present the many-to-many collective communication problem. This results in a new parameter δ (the degree of the communications graph) being added to the cost equations.

Most scientific applications [15, 23], tend to be iterative and have a persistent communication pattern which can be learned. A learning framework can record the number of messages sent from each processor and the size of each message, in each iteration. Based on this information, a strategy can be chosen at runtime using the cost equations presented in the paper. This operation can also be repeated periodically for a dynamic application with a varying communication pattern. We are currently working on developing such a framework.

Further research is needed to deal with incomplete 3d grids and hypercubes optimally. We also plan to improve the performance of our AAPC library for intermediate sized messages, where MPI's performance is currently better.

Our analysis of many-to-many personalized communication (MMPC) considers only the degree (δ) of the communications graph and not the actual destinations. This is suited for uniform many-to-many applications like *Neighbors* and the performance results validate our cost equations. In many cases certain intermediate nodes may get overloaded and violate the cost equations like the 3d FFT case with high variance. This can possibly be handled by reorganizing the virtual topology among the processors, especially using persistence-based "learning" strategies. Further, a more sophisticated cost framework which takes the actual destinations into account also needs to be designed.

Our current analysis and methods are topology independent. This is reasonable for a machine such as Lemieux (with a fat-tree topology). But for future machines such as Blue Gene/L, with relatively limited cross-section bandwidth, mapping of virtual topologies to real ones needs to be optimized.

Acknowledgments We would like to thank members of parallel programming laboratory including Gengbin Zheng, Orion Lawlor, Ramkumar Vadali, Chee Wai Lee and the staff of Pittsburgh Supercomputing Center for their assistance. This work was supported by the National Institutes of Health (NIH PHS 5 P41 RR05969-04) and the National Science Foundation (NSF NGS 0103645, NSF ITR 0121357).

References

[1] S. Bokhari. Multiphase complete exchange: a theoretical analysis. *IEEE Trans. on Computers*, 45(2), 1996.

[2] C. Christara, X. Ding, and K. Jackson. An efficient transposition algorithm for distributed memory clusters. In *13th Annual International Symposium on High Performance Computing Systems and Applications*, 1999.

[3] V. V. Dimakopoulos and N. J. Dimopoulos. Communications in binary fat trees. In *International Conference on Parallel and Distributed Computing Systems*, 1995.

[4] V. V. Dimakopoulos and N. J. Dimopoulos. A theory for total exchange in multidimensional interconnection networks. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):639–649, 1998.

[5] A. Gursoy. *Simplified Expression of Message Driven Programs and Quantification of Their Impact on Performance*. PhD thesis, University of Illinois at Urbana-Champaign, June 1994. Also, Technical Report UIUCDCS-R-94-1852.

[6] A. Gursoy and L. V. Kale. Simulating Message-Driven Programs. In *Proceedings of International Conference on Parallel Processing*, volume III, pages 223–230, August 1996.

[7] B. H. H. Juurlink, P. S. Rao, and J. F. Sibeyn. Worm-hole gossiping on meshes. In *Euro-Par, Vol. I*, pages 361–369, 1996.

[8] L. V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, Honolulu, Hawaii, April 1996.

[9] L. V. Kale and S. Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[10] C. C. Lam, C.-H. Huang, and P. Sadayappan. Optimal algorithms for all-to-all personalized communication on rings and two dimensional tori. *Journal of Parallel and Distributed Computing*, 43(1):3–13, 1997.

[11] Lemieux. http://www.psc.edu/machines/tcs/lemieux.html.

[12] P. K. McKinley, Y.-J. Tsai, and D. F. Robinson. A Survey of Collective Communication in Wormhole-Routed Massively Parallel Computers,. Technical Report MSU-CPS-94-35, 94.

[13] C. A. Moritz and M. Frank. Logpc: Modeling network contention in message-passing programs. In *Measurement and Modeling of Computer Systems*, pages 254–263, 1998.

[14] F. Petrini, W. chun Feng, S. Hoisie, A.and Coll, and E. Frachtenberg. The quadrics network: high-performance clustering technology. *IEEE Micro*, 22(1):46 –57, 2002.

[15] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. Namd: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[16] R. Ponnusamy, R. Thakur, A. Chourdary, and G. Fox. Scheduling regular and irregular communication patterns on the CM-5. In *Supercomputing*, pages 394–402, 1992.

[17] D. Scott. Efficient all-to-all communication patterns in hypercube and mesh topologies. In *Sixth Distributed Memory Computing Conference*, pages 398–403, 1991.

[18] Y. J. Suh and S. Yalamanchili. All-to-all communication with minimum start-up costs in 2d and 3d tori. 9(5), 1998.

[19] N. S. Sundar, D. N. Jayasimha, D. K. Panda, and P. Sadayappan. Hybrid algorithms for complete exchange in 2d meshes. In *International Conference on Supercomputing*, pages 181–188, 1996.

[20] A. Tam and C. Wang. Efficient scheduling of complete exchange on clusters. In *ISCA 13th International Conference On Parallel And Distributed Computing Systems*, August 2000.

[21] A. T. Tam and C.-L. Wang. Realistic communication model for parallel computing on cluster. In *1st IEEE Computer Society International Workshop on Cluster Computing*, December 1999.

[22] Thakur and Choudhary. All-to-all communication on meshes with wormhole routing. In *IPPS: 8th International Parallel Processing Symposium*. IEEE Computer Society Press, 1994.

[23] M. S. Warren and J. K. Salmon. Astrophysical n-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing 92*, Nov. 1992.