Automatic Cluster Parallelization and Minimizing Communication via Selective Data Replication

Sanket Tavarageri, Benoît Meister, Muthu Baskaran, Benoît Pradelle, Tom Henretty, Athanasios Konstantinidis, Ann Johnson, Richard Lethin
Reservoir Labs, 632 Broadway, New York, NY 10012
Email: \{tavarageri,meister,baskaran,pradelle,henretty,konstantinidis,johnson,lethin\}@reservoir.com

Abstract—The technology scaling has initiated two distinct trends that are likely to continue into future: first, the increased parallelism in hardware and second, the increasing performance and energy cost of communication relative to computation. Both of the above trends call for development of compiler and runtime systems to automatically parallelize programs and reduce communication in parallel computations to achieve the desired high performance in an energy-efficient fashion.

In this paper, we propose an auto-parallelization method and runtime system that auto-paralleлизes loop-nests to clusters and, a novel communication avoidance method that reduces data movement between processors. Communication minimization is achieved via data replication: data is replicated so that a larger share of the whole data set may be mapped to a processor and hence, non-local memory accesses reduced. Experiments on a number of benchmarks show the effectiveness of the approach.

I. INTRODUCTION

The increasing complexity and heterogeneity of supercomputers as we move beyond petaflop systems has called for an urgent development of programming and runtime systems that automatically deal with the complexity and at the same time run computations in a way that is efficient both from performance and energy considerations. The main challenges to address in the context of parallel computers, inter alia, are: effective parallelization and communication management between parallel processors. As the cost of communication has increased significantly relative to the cost of computation [3, 15], it has become crucial that new techniques be developed that minimize communication in parallel computations. It is further desirable that the techniques be general and adapt to constraints imposed by system resource availability (such as memory size).

In this paper, we propose an auto-parallelization method that separates the concern of finding parallelism in the computation to that of movement of data in the parallel computer: the compiler parallelizes code and inserts virtual memory instructions that specify data that will be consumed and produced by the parallel task. The runtime is responsible for initiating and orchestrating communication by being cognizant of the underlying message passing mechanism.

We also present a novel communication minimizing scheme that trades off communication to the use of memory in a flexible fashion. The approach stores redundant copies of selected data structures on the distributed memory of the system so that accesses to them can be serviced locally and reduce inter-node communication. The requests for data are routed to the nearest processor and coherence among multiple copies of data is automatically managed.

The proposed communication-minimizing approach is general in that it is not specific to any particular algorithm. Compiler-generated inspectors guide the runtime system in making decisions with regard to how much additional memory to use and what to store in that additional memory.

This paper makes the following contributions:

- An integrated compiler and runtime system for cluster auto-parallelization that decouples the concerns of computation-placement and data-placement and thus facilitates dealing with the system complexity effectively.
- An algorithm-agnostic approach to communication minimization by selectively replicating data.
- An analytical investigation of the relation between program characteristics, viz., the number of reads and writes, to the best data replication arrangement.
- Experimental evaluation of the effectiveness of the communication minimizing scheme.

II. AUTOMATIC CLUSTER PARALLELIZATION

The R-Stream compiler [13] is a source-to-source compiler for automatic parallelization of sequential programs. It accepts loop nests written in C and produces parallelized codes for multiple targets, including multi-core machines, GPUs, and FPGAs. We have extended R-Stream to perform cluster parallelization. The R-Stream compiler uses the polyhedral model [10] for program analysis and transformation. It implements high performance techniques that enhance data locality and perform parallelization (e.g., [12, 17]).

The generated cluster-parallel programs have the SPMD (Single Program Multiple Data) form. R-Stream aggregates loop iterations into tasks as part of its parallelization process. The aggregation process uses the tiling program transformation [11, 18]. Data communication between processors is performed at the boundaries of these tasks.

Communication operations are abstracted as logical DMA (Direct Memory Access) primitives — each task issues logical DMA GETs to fetch data needed for computation and PUTs to store live-out data produced by the task. The logical DMA operations are in turn implemented as an R-Stream runtime layer functionality using the Global Arrays toolkit [14]. Global Arrays (GAs) provide a global address space for creating and accessing arrays.

We illustrate cluster-parallelization in R-Stream using an...
int i; 
for (i=0; i<N; i++) {
    A[i] = B[i] + 1;
}

Fig. 1: Input code

example. Consider the loop shown in Figure 1. It adds a constant — 1 to N elements of array B and stores the result in array A. The R-Stream compiler, for this input and a 4-node cluster, produces parallelized code shown in Figure 2. The computation is partitioned such that each processor increments $\frac{N}{4}$ elements of array B. A DMA GET instruction is issued for the data required by the task and the data written in the loop are stored using a DMA PUT operation at the end.

The DMA formation phase in R-Stream emits efficient logical DMA operations: whenever the data to be read and written have contiguous portions in them, data movement will be orchestrated in such a way that there will be a single DMA call for a contiguous segment of data.

We note that the R-Stream compiler can generate two-level parallelized code — one for inter-node execution (which is the focus of the paper) and one for intra-node (OpenMP being one of the several programming models and runtimes available for intra-node parallelization). Since cluster parallelization and communication minimization between nodes are the focus of this paper, the example presented shows one-level (inter-node) parallelized code. The subsequent discussion and experimental evaluation are also focused on cluster-centric techniques.

int PROC = r_nodeid();  
float A[N/4];  
float B[N/4];  
r_dma_get(B_id, (N/4)*PROC, B, 1, 1, N/4);  
int i;  
for (i = 0; i < N/4; i++) {
    A[i] = B[i] + 1;
}

r_dma_put(A, A_id, (N/4)*PROC, 1, 1, N/4);

Fig. 2: Cluster-parallelized output code

III. COMMUNICATION MINIMIZATION VIA DATA REPLICATION

Global arrays [14] reside in a Partitioned Global Address Space (PGAS) of a cluster. They are distributed among the memories of participating nodes. A chief advantage of global arrays is that the parallel program is highly scalable with respect to memory usage — the parallel program can handle problem sizes whose data structures can be allocated on the collective memory space of the system.

When additional memory beyond the minimal amount required to allocate all arrays used by a computation is available, we use the excess memory capacity at our disposal to reduce communication between nodes.

Figure 3 shows the distribution of a global array of size N among four nodes. A quarter of the array is resident on each node. Replicating an array has the following advantages:

1) It increases the fraction of the global array stored on any one node, thereby increasing the number of DMA GETs serviced locally. Consequently, non-local (remote) memory accesses are proportionately reduced and communication between nodes is minimized for DMA GETs.
2) When a requested array cell through DMA GET is not locally available, the desired array element can be retrieved from a copy that is closer to the requested node, hence reducing latency of the message. Steering request for a data element to different servers also helps distribute communication traffic on the system and reduces the possibility of performance hot-spots.

Fig. 3: Distribution of a Global Array

Figure 4 shows a duplicated global array. Node 1 and 2 hold one copy of the array; node 3 and 4 another. Each node is now home to half of the array (as opposed to a quarter earlier) and DMA GET requests to half of the array are serviced from the local memory. When an array cell is not available in local memory, the DMA GET request is directed to the closest node that holds the data. The closest node is defined in terms of MPI ranks. But, it could be modified to incorporate network topology information. The diagram shows that an array element $i$ can be obtained either from node 1 or 3.

Fig. 4: Replicated Global Array; A DMA GET can access any copy of the array

Fig. 5: Replicated Global Array; A DMA PUT updates all copies of the array

To maintain coherence among different copies of data, when a DMA PUT is issued for an array cell, the write is propagated to all copies. Figure 5 depicts the scenario when array element at index $i$ is written. The updated value is sent to both the locations that are replicas of cell $i$.

Replication of data thus decreases remote memory accesses for DMA GETs, but increases remote memory accesses for DMA PUTs. Therefore, the replication scheme is beneficial
when DMA GET data volume is larger than that of DMA PUTs. Equivalently, the overall communication is reduced when the number of reads is greater than the number of writes. In the next section, we analyze the interplay between the read-to-write ratio and data movement on the cluster.

We note that the number of reads being higher than the number of writes in programs is the common case. Hence, the communication minimization method developed here is widely applicable.

IV. DATA REPLICATION CONSIDERATIONS

The communication minimizing configuration — which arrays to replicate and by how much — depends on the amount of communication generated by DMA GETs and PUTs in the computation. The compiler inserts inspector codes which inform the runtime system of the read and write characteristics of the program, and the runtime then accordingly makes decisions on data replication.

A. Compiler Generated Inspector Codes

The R-Stream compiler inserts inspectors before the actual computation code that call dummy virtual DMA functions. The runtime, on a per-array basis, keeps count of the total number of elements read through GETs and written through PUTs on each node. It then accumulates counts on all nodes and calculates the grand total of reads — R and, writes — W on a per-array basis.

B. Best Data Replication Factor

We would like to derive data replication factor α that reduces data movement between nodes. We first reason about the expected number of remote memory accesses without data replication and then the number of array elements communicated between nodes after the array is replicated α times. If the array size originally is N, after data replication its size becomes αN and the array is distributed in equal chunks on all nodes in either case. Data replication factors are determined for each array separately.

Let the number of nodes in the system be N and the number of array elements read by node i through DMA GETs be ri and written via DMA PUT be wi. Therefore, the total number of reads R by all nodes is: R = \sum_{i=1}^{N} ri. Similarly, the total number of writes W is: W = \sum_{i=1}^{N} wi.

Before data replication: The probability that a requested array cell is mapped locally is proportional to \frac{1}{N} and in fact it is \frac{1}{N} if the accesses to the array are uniformly random. However, when the compiler has performed optimized computation placement, we can expect the data references to not be fully random, and the array access behavior favors local portions of the array. To model data access characteristics of real applications, we introduce a correction factor β such that the fraction of local accesses is \frac{β}{N}. We note that when β = 1, the probability defaults to uniformly random accesses and when β > 1, it characterizes an optimized computation placement. Hence, the expected number of reads to local memory on node i is: \frac{R}{N}. The total number of reads to local memory on the system thus is: \sum_{i=1}^{N} \frac{R}{N} = \frac{R}{N}. The cumulative number of writes to local memory in a like fashion is \frac{W}{N}.

Consequently, the number of local memory accesses (reads and writes combined) is: \frac{R + W}{N}. The number of remote memory accesses is: \frac{R}{N} - \frac{R + W}{N} = \frac{W}{N}.

After data replication: When an array is replicated α number of times, \frac{1}{N} fraction of the array is assigned to any given node. Therefore, the probability that a requested array cell can be found on the same node is proportional to \frac{α}{N} and using the correction factor β, it is \frac{αβ}{N}. Consequently, the expected number of local reads across the system is \frac{αβR}{N}. The expected number of remote reads is: \frac{R}{N} - \frac{αβR}{N} = \frac{R(1 - αβ)}{N}.

When a write to an array cell is performed by node i, the write will be broadcast to α copies of the array. Hence the number of writes made by i will be αw_i. Let us consider a single application-level write which translates to α runtime-level writes to α different replicas of the array. Of the α writes, at most one write can be local, the others will be remote: more than one copy of any array cell will not be mapped to the same node. The probability that the one write will be local is \frac{αβ}{N}. Therefore, the expected number of local writes is: \frac{αβ}{N}w_i. The total number of local writes is \sum_{i=1}^{N} \frac{αβ}{N}w_i = \frac{αβW}{N}. The total number of remote writes is: \frac{αW}{N} - \frac{αβαβ}{N}W = \frac{αW(1 - αβ)}{N}.

The number of remote reads and writes combined will be:

\frac{R(1 - αβ)}{N} + \frac{αW(1 - αβ)}{N}.

Data replication condition: We want to minimize the number of remote memory accesses. It is observed that after data replication, the number of remote reads is decreased while the number of remote writes is increased. And, we would like the total number of remote memory accesses — reads and writes combined after data replication to be less than that before data replication.

If W > 0, and for remote memory accesses after data replication to be less than before data replication, we must have:

\frac{R(N - αβ)}{N} + \frac{αW(N - αβ)}{N} < \frac{(R + W)(N - αβ)}{N}.

\frac{R(N - αβ)}{N} + \frac{αW(N - αβ)}{N} < \frac{(R + W)(N - αβ)}{N}.

\frac{αW(N - αβ)}{N} < \frac{R(N - β)}{N}.

\frac{αβW(N - αβ)}{N} < \frac{R}{W}.

\frac{αβW(N - αβ)}{N} < \frac{R}{W}.

Inequality (1) indicates that if the read-to-write ratio is greater than \frac{R}{W} - 1, data replication is probabilistically guaranteed to reduce communication. We observe that when the probability of a data reference accessing any part of the global array is the same (i.e., when β = 1), the required read-to-write ratio is N - 1. As the probability changes to favor more of locally mapped portions of the global array (β > 1), the minimum read-to-write ratio decreases to a smaller value.

Remote memory accesses: The number of remote memory accesses reduced — difference between the number of remote accesses before and after data replication is as follows.

\frac{(R + W)(N - αβ)}{N} - \frac{R(N - αβ)}{N} = \frac{RαβW}{N}.
\[
\frac{(\alpha - 1)\beta R}{N} - \frac{(\alpha - 1)W(N - \beta)}{N} = (\alpha - 1)\left(\frac{\beta(R + W)}{N} - W\right)
\] (2)

Expression (2) represents reduction in remote memory accesses. We note that for all values of \(\beta\), as the data replication factor \(\alpha\) increases, reduction in remote memory accesses also increases. Equivalently, remote memory accesses are reduced.

C. Overall Algorithm for Data Replication

Given array sizes, read-to-write ratios, and the memory capacity of the system, there can be multiple ways of replicating arrays that use up the available memory. But, we would want to find an optimal data replication regimen — assignment of values to \(\alpha\) s, that results in minimal remote memory accesses.

An ILP Formulation: We want to maximize the decrease in remote memory accesses, subject to the constraints that the memory capacity is not exceeded on each node, while at least one copy of each array is maintained in the system. Given \(k\) arrays with array sizes \(s_j\) and per-node memory capacity \(C\), we formulate the problem of finding optimal per-array replication factors \(\alpha_j\)'s as an Integer Linear Programming (ILP) problem. Expression (2) gives the reduction in remote accesses for a single array and we want to maximize the sum of remote memory reductions across all arrays.

Maximize \[
\sum_{j=1}^{k} (\alpha_j - 1)\left(\frac{\beta_j(R_j + W_j)}{N} - W_j\right)
\]
subject to: \[
\sum_{j=1}^{k} \alpha_j s_j \leq C N, \quad \text{and} \quad \alpha_j > 1
\]

Heuristic: Even though the ILP formulation would give an optimal solution to the problem of replication assignment, solving the ILP problem would be expensive at runtime. Therefore, we propose a heuristic based on the premise that given a choice to replicate either array \(A\) or array \(B\), it would be more beneficial to replicate the array with a higher read-to-write ratio. (Array sizes, local access probabilities would not make that premise true in general).

Algorithm 1 presents the data replication strategy. The replication decisions are made starting with the array that has the highest read-to-write ratio: the higher the number of reads relative to the number writes, greater is the potential to save communication via replication. We can replicate an array meaningfully only up to the total number of nodes: we can at most have one copy of the array at each node.

V. EXPERIMENTAL EVALUATION

We have implemented the auto-parallelization and communication minimizing algorithms in the R-Stream compiler. In this section, we describe experiments conducted to evaluate communication minimization achieved by the integrated compiler and runtime scheme presented.

A. Set up

Table I lists the benchmark programs and problem sizes used for experiments. The corcol code computes a correlation matrix. The gemver benchmark does vector multiplication and matrix addition, while doltgen is a multi-resolution analysis kernel. The planck and hydro are from the Livermore loops. planck derives Planckian distribution and hydro is a 2-D explicit hydrodynamics fragment. The stencil — Reverse Time Migration (RTM) is used in a finite difference discretization of the wave equation when performing seismic imaging. The amf and covar are adaptive matched filtering and covariance estimation components of the space-time adaptive processing (STAP) [9] application domain that is part of the DARPA PERFECT benchmark suite [2].

The codes were cluster-parallelized using R-Stream and were compiled with PathScale (tm) Compiler Suite: Version 4.0.10 and -O3 flag. The resulting binaries were run on 8 nodes of a cluster. Each node is made up of AMD Opteron(TM) Processor model 6272 processors. One process was launched per node as inter-node communication optimization is the focus of this work. For each benchmark, experimental data — number of local and remote memory accesses, running time are gathered for two program variants: 1) auto-parallelized codes 2) auto-parallelized and communication minimized programs (marked with a ‘:c’ suffix in the ensuing graphs).

The communication optimization is achieved by selective data replication. The inspectors inserted by the R-Stream compiler drive data replication decisions (§IV). The inspection phase computes the read-to-write ratio to each array used in the application. In our experiments, we experimented with different threshold \(R_{W_j}\) ratios and Table I reports the minimum read-to-write ratio that was required for each benchmark program to achieve maximum reduction in data movement. We note that the minimum \(R_{W_j}\) value necessary is small: 2

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Problem size</th>
<th>(\frac{R}{W_j})</th>
</tr>
</thead>
<tbody>
<tr>
<td>corcol (c)</td>
<td>3000 X 3000</td>
<td>2</td>
</tr>
<tr>
<td>gemver (g)</td>
<td>3000 X 3000</td>
<td>3</td>
</tr>
<tr>
<td>doltgen (d)</td>
<td>50 X 50 X 50</td>
<td>2</td>
</tr>
<tr>
<td>planck (p)</td>
<td>5000</td>
<td>2</td>
</tr>
<tr>
<td>hydro (h)</td>
<td>2000 X 2000</td>
<td>3</td>
</tr>
<tr>
<td>RTM (r)</td>
<td>264 X 264 X 264</td>
<td>2</td>
</tr>
<tr>
<td>amf (a)</td>
<td>4 X 512 X 32</td>
<td>2</td>
</tr>
<tr>
<td>covar (v)</td>
<td>4 X 512 X 32</td>
<td>4</td>
</tr>
</tbody>
</table>
for a majority of benchmarks and at most 4. Therefore, the communication avoidance method is applicable to a wide range of applications.

B. Results

Figures 6 and 7 show data movement statistics of a) parallel programs and b) parallel plus communication minimized programs. Data movement between nodes is triggered when a virtual DMA GET or a PUT accesses parts of the Global Array that are resident on other nodes. The number of bytes transferred for communication-minimized benchmarks are normalized with respect to that of parallelized but not communication optimized programs.

The maximum data movement reduction is achieved in planck code: the number of remote memory accesses are decreased by 47%. Out of five arrays in planck (p), three are read-only arrays and they are replicated. It is the main reason for significant reduction in remote memory accesses. The corcol (c) benchmark uses 8 global arrays, of which 4 are replicated: two are read-only arrays, while two others are both read- and write- arrays. It finds its remote memory accesses reduced by 17%. The geometric mean of reduction in the number of remote memory operations across benchmarks is 15.5% (the last two bars marked m and m: c. m stands for mean).

The number of local accesses for benchmarks are depicted in Figure 6. The trends are exact opposite of remote accesses — as remote accesses are lowered, local accesses are proportionately enhanced. On average (geometric mean), the number of local accesses is 1.93X higher with communication minimized programs. We observe that the selective data replication approach converts energy-hungry remote memory accesses to inexpensive local accesses.

Figure 8 shows the normalized running times of applications. The break-down of execution time is also depicted: times spent in performing virtual DMA GETs, DMA PUTs, computation and inspection are graphed. We observe that the inspection time is a very tiny fraction of the total time. In five out of eight benchmarks — gemver (g), hydro (h), and RTM (r), amf (a), and covar (v), a majority of time is expended in communicating — in DMA GETs and PUTs. The execution time of communication avoiding codes is 1.6% less than that of just parallel codes on average (geo-mean).

The energy cost of communication relative to computation is orders of magnitude higher on current systems and the trend is expected worsen as technology scales [15, 5]. Table II shows the intra-node and inter-node memory access costs. The remote memory accesses are 15 times more expensive compared to local memory accesses. Using these representative communication costs, we estimate reduction in communication energy from the communication minimization techniques presented in the paper. Figure 9 depicts the energy expended by communication-avoiding codes in accessing local and remote memories when compared to non-optimized codes. The communication energy reduction is 44% in planck (p) benchmark while it is 24% in RTM (r). The communication-avoiding techniques on average (geometric mean) reduce communication energy by 14%. We note that remote memory access energy dominates the total communication energy budget. Further, the proportion of energy spent in accessing remote memory is lower in communication-minimizing codes because remote memory accesses have been traded off for energy-efficient local memory accesses.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Local memory cost</th>
<th>Remote memory cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4pJ/byte</td>
<td>60pJ/byte</td>
</tr>
</tbody>
</table>

VI. RELATED WORK

There has been a significant amount of research in the realm of automatic cluster parallelization [1, 4, 6]. Amarasinghe and Lam [1] present compiler algorithms using the polyhedral model for generation of required communication — receive and send instructions for a given computation and data distribution. More recently Dathathri, Bondhugula et al. [4, 6] have proposed techniques to reduce inefficiencies in communication generation schemes of earlier works.

We take a different approach to communication generation in this paper: communication responsibility is split between the compiler and the runtime. The compiler identifies data that
the computation consumes and produces. The runtime handles
the job of placement of data and communication between
processors. This method, unlike compiler-only methods, does
not bake communication into generated code, instead affords
flexibility to dynamically place and communicate data in a
resource-aware manner. For example, the decision to replicate
data to a varying degree based on memory availability of the
prevailing execution environment could not be easily accom-
plished in a fully compiler-generated communication scheme.

Communication minimization has also received a lot of
attention from the research community. The communication
avoiding algorithms for various numerical algebra problems
— such as matrix multiplication, LU decomposition [16, 7, 8]
have been developed and operate in 2.5D processor grids (they
are 3 dimensional grids and one of the dimensions is of a
constant size, hence the name 2.5D). The idea is to trade off
higher memory use (via data replication) for communication.
The algorithms replicate either read-only data or reduction
arrays and are applicable only for certain processor grid config-
urations, namely 2.5D. The automatic data replication strategy
for communication avoidance presented in this paper is based
on the same concept of trading memory to communication,
but is more general in three significant ways: 1) it is not
algorithm-specific; 2) even write data may be replicated and
data consistency is automatically maintained; 3) it is applicable
to all processor grid configurations.

VII. CONCLUSION

The vast parallelism available in today’s systems requires
that effective compiler technology be developed to parallelize
programs. At the same time, energy has become a first-
class design constraint in the design of computing systems
and the data movement cost is projected to be orders of
magnitude higher than that of computation cost. Consequently,
reducing data movement will reduce energy expenditure and
thus contribute substantially to easing the energy envelope.

In this paper, we presented an integrated compiler and
runtime approach to auto-parallelization of codes for clusters
and a communication optimization method that avoids com-
munication via selective data replication. The runtime makes
optimal decisions on data replication with input from the
compiler. Our experiments on a set of benchmarks indicate
that the proposed techniques reduce data movement between
processors and associated energy costs significantly – up to
44% in some applications.

ACKNOWLEDGEMENT

We thank the U.S. Department of Energy and DARPA for
their funding and Pacific Northwest National Laboratory for
access to their Olympus computing cluster.

REFERENCES

optimization and code generation for distributed memory
Conference on Programming Language Design and Im-
[2] K. Barker, T. Benson, D. Campbell, D. Ediger,
R. Gioiosa, A. Hoisie, D. Kerbyson, J. Manzano, A. Mar-
quez, L. Song, N. Tallent, and A. Tumeo. PERFECT
(Power Efficiency Revolution For Embedded Computing
Technologies) Benchmark Suite Manual. Pacific North-
west National Laboratory and Georgia Tech Research
PERFECT/.
study: Technology challenges in achieving exascale sys-
tems. Defense Advanced Research Projects Agency In-
formation Processing Techniques Office (DARPA IPTO),
distributed-memory parallel architectures. In Interna-
tional Conference for High Performance Computing, Net-
working, Storage and Analysis (SC), pages 1–12. IEEE,
2013.
2013.
jugula. Generating efficient data movement code for
heterogeneous architectures with distributed-memory. In
22nd International Conference on Parallel Architectures


