Global Partitioning of Parallel Loops and Data Arrays for Caches and Distributed Memory in Multiprocessors

Rajeev Barua*
MIT Laboratory for Computer Science
545 Technology Square, Room NE43-633
Cambridge, Massachusetts 02139

1 Introduction

The problem of loop and data partitioning for distributed memory multiprocessors with global address spaces has been studied by many researchers.[1] is one of them, see that paper for more. The goal of *loop partitioning* for applications with nested loops that access data arrays is to divide the iteration space among the processors to get maximum reuse of data in the cache, subject to the constraint of having good load balance. For architectures where non-local memory references are more expensive than local memory references, the goal of *data partitioning* is to place data in the memory where it is most likely to be accessed by the local processor. Data partitioning tiles the data space and places the individual data tiles in the memory modules of the processing nodes.

In this paper we will present a solution to the problem of determining loop and data partitions automatically for programs with multiple loops and data arrays. We assume that parallelism in the source program is specified using parallel **do** loops. This can either be done by a programmer, or by a previous dependence analysis and parallelization. The full version of this work is in [2].

We have applied our algorithm to cache-coherent multiprocessors with physically distributed memory. We introduce a cost model that will estimate the cost of executing a loop given the loop partitions and the partitions of data arrays accessed by the loop. This cost model is based on architectural parameters such as the cost of local and remote cache misses. The cost model is used to drive an iterative solution to the global problem.

We note that the demands of cache and data locality often conflict. The algorithm itself uses an iterative improvement method to arrive at a solution. We initially begin with a cache optimized solution as demanded in [1], and the heuristic method then determines the best data partitions given the loop partitions. Then the loop partitions are re-evaluated given the new data partitions. These forward and back phases of the algorithm are run until some threshold number of iterations, and the best solution found until then, as determined by the cost model, is used.

2 Loop Partitioning Overview

This section gives a brief summary of the method for loop partitioning to increase cache reuse given in [1]. We use this method as the starting point for loop and data partitioning. The method handles programs with loop nests where the array index expressions are affine functions of the loop variables. In other words, the index function can be expressed as,

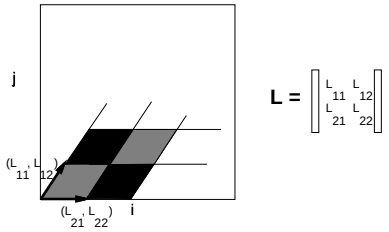

Figure 1: Iteration space partitioning is completely specified by the tile at the origin.

$$\vec{g}(\vec{i}) \quad = \quad \vec{i}\mathbf{G} + \vec{a} \tag{1}$$

where $\mathbf{G}$ is a $l \times d$ matrix with integer entries and $\vec{a}$ is an integer constant vector of length d, termed the *offset vector*. Thus accesses of the form A[2i+j,100-i] and A[j] are handled, but not A[i^2], where i,j are nested loop induction variables.

A loop partition $\mathbf{L}$ is defined by a hyperparallelepiped at the origin as pictured in Figure 1. The *footprint* of an iteration tile $\mathbf{L}$ with respect to an array reference is the set of points in the data space accessed by the loop tile as a result of the reference. This footprint is given by $\mathbf{LG}$.

[1] shows how $\mathbf{L}$ can be chosen to minimize the number of cache misses. In the cost model we also refer to the data partition $\mathbf{D}$. $\mathbf{D}$ represents how the data space is tiled. This is represented as a tile at the origin of the data space just like $\mathbf{L}$ is represented as a tile at the origin of the iteration space. An array reference in a loop will have good locality when $\mathbf{LG} = \mathbf{D}$.

3 The Cost Model

In order to evaluate the combined cache and data locality of loop and data partitions we use a cost function to compare different solutions. This function takes, as arguments, a loop partition, data partitions for each array accessed in the loop, and architectural parameters that determine the relative cost of cache misses and remote memory accesses. It returns an estimation of the cost of array references for the loop.

To begin with, we can express the cost due to memory references in terms of the architectural parameters in the following equation:

*E-mail: {barua}@lcs.mit.edu. The research reported on herein was supervised and co-researched by Professor Anant Agarwal and Dr. David Kranz. It was funded in part by NSF grant # MIP-9012773, in part by ARPA contract # N00014-91-J-1698, and in part by a NSF Presidential Young Investigator Award.

$$T_{total\_access} = T_R(n_{remote}) + T_L(n_{local}) + T_C(n_{cache})$$

where T_R, T_L, T_C are the remote, local and cache memory access times respectively, and $n_{remote}, n_{local}, n_{cache}$ are the number of references that result in hits to remote memory, local memory and cache memory. These depend on the loop and data partitions.

The full thesis [2] shows how we can express the above simple formula in terms of the partitioning variables defined in [1]. The derivation uses a series of intermediate functions, and is somewhat involved. An exact formula is very hard to compute, but we have made some assumptions which are almost always true in practice, giving a relatively simple formula.

4 The Multiple Loops Heuristic Method

This method uses a commonly used natural representation of a program with many loops accessing many arrays, namely, a bipartite graph G = (V_l, V_d, E). We picture the loops as a set of nodes V_l on the left hand side, and the data arrays as a set of nodes V_d on the right. An edge e∈E between a loop and array node is present if and only if the loop accesses the array.

4.1 Iterative Method

The basic method is the following. We begin with an initial loop partition arrived at by the single loop optimization method described in Section 2. Then we follow an iterative improvement method with each iteration having two phases: the first (forward) phase finds the best data partitions given loop partitions, and the second (back) phase re-evaluates the values of the loop partitions given the data partitions just determined.

Specifically, in the forward phase we set the data partition of each array to be one of the data partitions induced by the loops accessing it. The choice is to pick the data partition induced by the largest loop accessing it. There are some details and changes to this basic strategy which we discuss in [2]. In the back phase we set the loop partition of each loop to be one of the loop partitions induced by the arrays accessed by the loop. We use the cost model to evaluate the alternative loop partitions and pick the one with the minimum cost.

These forward and backward phases are repeated, each time using the cost model to determine the estimated array reference cost for the current partitions. After some number of iterations, the best partition found so far is picked as the final partition. A parameter to be determined is how many iterations should be used. [2] shows that the length of the longest acyclic path in the graph is a reasonable bound. In practice this heuristic seems to perform quite well.

5 Results

We have implemented the algorithm described in this paper as part of our compiler for the MIT Alewife machine. The Alewife machine implements a shared global address space with distributed physical memory and coherent caches. The nodes contain slightly modified SPARC processors and are configured in a 2-dimensional mesh network. The approximate Alewife latencies are: 2 cycle cache hit, 11 cycle local memory hit, 40 cycle remote access assuming no contention. The last number will be larger for large machine configurations.

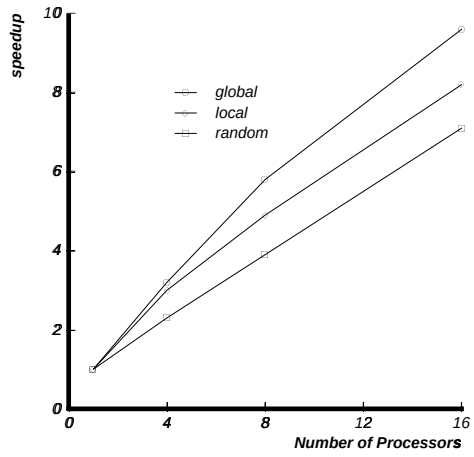

Figure 2: Speedup for conduct. Problem size 153 x 133, double precision.

As an example, we ran the *conduct* routine from the SIMPLE application on the Alewife machine simulator. It has 20 loop nests, and 20 arrays. A static partition for the data is used. We ran the *conduct* code using three different data partitions:

global This is obtained by using the algorithm described in this paper.

local This is the result of partitioning as in [1] for cache locality, looking at one loop at a time.

random This partition just allocates the data across processors with no regard for data locality.

Figure 2 shows the speedups obtained for the global, local, and random partitions. We see that the global partitioning heuristic provides a significant increase in performance over local, which itself is significantly faster than using a random data partition. We will have more data for a final paper soon, including a larger problem size and larger number of processors, as well as results of varying remote memory latency.

References

[1] Anant Agarwal, David Kranz, and Venkat Natarajan. Automatic Partitioning of Parallel Loops for Cache-Coherent Multiprocessors. In *22nd International Conference on Parallel Processing*, St. Charles, IL, August 1993. IEEE. A version of this paper appears as MIT/LCS TM-481, December 1992.

[2] Rajeev Barua. Global partitioning of parallel loops and data arrays for caches and distributed memory in multiprocessors. Master's thesis, MIT, Department of Electrical Eng and Comp Sci., May 1994.