Kernel-Control Parallel Versus Data Parallel: A Technical Comparison

Extended Abstract

Terrence W. Pratt
Center of Excellence in Space Data and Information Sciences (CESDIS)
Code 930.5
NASA Goddard Space Flight Center
Greenbelt, MD 20771
e-mail: pratt@cesdis.gsfc.nasa.gov

The *dusty deck problem* is the problem of how to automatically transform large existing serial/vector codes into a form suitable for efficient execution on new parallel architectures. *Kernel-control parallel* (KCP) methods provide a promising approach to solving the dusty deck problem for MIMD distributed memory machines. This paper is a status report on a project to develop this technology into a full-scale prototype for running large Fortran 77 codes on the Intel iPSC/860. Space limits preclude a full description of the methods. Instead, we sketch the approach and then use a series of questions to explore some of the key differences between this new technology and existing data parallel methods, which are similar but better known.

Background

Intel, Cray Research, TMC, and other vendors have announced their intention to base the next generation of software systems for MIMD distributed memory parallel machines on some variant of *data parallel* software technology, as exemplified by the emerging definitions of High Performance Fortran (HPF) [1], Vienna Fortran [2], and Fortran-D [3]. The software goal is to allow existing serial/vector codes (''dusty decks'') to be ported to these parallel machines relatively easily, although not automatically. In general, data parallel technology requires that existing codes be restructured and rewritten.

Using data parallel technology, the user supplies directives (usually in a Fortran extension) to specify how arrays should be distributed and aligned during program execution. The compiler uses these annotations to determine the appropriate placement of data and computation during program execution. It then generates the parallel program and links it to a run-time library that provides communications and other services.

Execution is based on the *loosely synchronous SPMD (single program, multiple data)* model, in which each processor executes the same basic program, with some parts of the computation replicated on all nodes and some parts assigned uniquely to individual nodes. At periodic synchronization points, nodes exchange data and coordinate their patterns of data access.

What is Kernel-Control Parallel?

Kernel-control parallel (KCP) is the name for a collection of methods that provide a new software technology for attacking the dusty deck problem. The approach is based both on a new conceptual framework and on new software solutions to the difficult problems posed by how languages like Fortran 77 are used in practice. The conceptual framework is based on a theoretical model called *kernel-control decomposition* developed in the 1970s for other purposes [4]. The concepts have been implemented in a series of software prototypes. Solution of performance problems uncovered by the prototypes has led to several key additions to the conceptual framework.

The kernel-control parallel approach is based on four major concepts:

1. *Kernel-control decomposition.* The theory of kernel-control decomposition provides the basis for a new method for deriving a parallel program from a sequential program. The basic technique is this. The source program variables are partitioned into *kernel variables* (usually real arrays and files) and *control variables* (usually scalars and integer/character arrays). Within the executable code, *kernel segments* are identified. A kernel segment is a single-entry, single-exit block of code that contains references to kernel variables. All references to a kernel variable that involve storing or fetching the value of the variable must fall within some kernel segment. It must be possible to accurately identify the IN and OUT kernel variables that are used and set within each kernel segment. Kernel segments may be of arbitrary ''grain size'' — from single statements to whole subroutine bodies. Each kernel segment represents a *schedulable execution unit* and is individually scheduled on a (possibly different) node each time it is executed. The IN and OUT variables for each segment are individually tracked during execution.

The remainder of the program, outside of the kernel segments, is termed the *control computation*. Control variables may be used in kernel segments or within the control computation, but it is undesirable to assign a new value to a control variable within a kernel segment (for reasons explained below). Control variables and the control computation are replicated on every node during execution.

Generation of a parallel program from a sequential program using kernel-control decomposition does not appear to require sophisticated compiler analysis of the source text. To date the prototypes have used manual source annotation to identify kernel variables, kernel segments, and IN/OUT data sets. Experience indicates that automation of the process is feasible using existing compiler technology.

2. *Asynchronous SPMD.* The execution model used in KCP technology is a variant of the SPMD model, characterized by a total lack of process synchronization during execution. All nodes execute the same basic program (one process per node), as in traditional SPMD, with the control computation replicated on all nodes and kernel segments executed by single nodes. Execution begins with all nodes starting at the main program and proceeding asynchronously down the control path, just as though each were executing the original serial program. When a kernel segment is reached, one node executes that segment and all others skip the segment.

During execution, nodes wait for only two reasons: (1) to receive any remote IN data values for kernel segments that they must execute and (2) to receive broadcasts of new values for control variables that are set within kernel segments. Both these types of communication represent true data dependences within the original program — data values that must be known in order for the receiving node to continue execution. There is no added communication necessary to represent control dependences, anti-dependences, or output dependences in the original code. Thus there is never any waiting during execution other than to satisfy true data dependences.

3. *Multidef variables.* In our KCP implementation, all kernel variables (including array elements) are implemented as *multidef variables.* A multidef variable is a variable that may exist in multiple "definitions" simultaneously during parallel execution. Each "definition" contains a different value, stored during a different execution of a kernel segment. Attached to the value is a tag indicating the definition point of the value (e.g., execution of the 724th kernel segment). A new definition of a multidef variable is created when that variable is an OUT variable of a kernel segment being executed on a node. The definition continues to exist until all uses of that definition (local or remote) are satisfied. When a node needs a local copy of the value of a multidef variable stored on another node, it fetches the value by using both the variable name and the *definition tag.* Thus instead of requesting the "value of X", a node requests "the value of X with definition tag 724". These semantics are similar to those used in *single assignment* languages, where variables may be assigned values only once.

Multidef variables provide the effect of shared variables. Both local and remote access is supported, and consistency of global values and local copies can be guaranteed. However, no synchronization is required either for fetch or store operations (no barriers, monitors, rendezvous, locks, critical regions, etc.). A store operation creates a new definition; a fetch specifies which definition is being accessed. The tradeoff lies in the additional run-time overhead of implementing these semantics in return for avoiding all synchronization.

4. *Scouting.* When a node is scheduled to execute a kernel segment, some of the IN values of kernel variables usually will be stored remotely on other nodes. Scouting provides an optimal way to *prefetch* these IN values. A *scout* is a special parallel process used to look ahead in the computation to determine what data transfers will be required and to start these data transfers as early as possible, preferably (the theoretical optimum point) immediately after the data values have been computed and stored on the remote node.

In the KCP model, a scout is implemented as a *scout node.* The scout node executes the same application program as all other nodes, proceeding asynchronously down the same control path. However, the scout node never executes any kernel segments. Hence, the scout node never pauses except to receive the broadcasts of control values that are computed within kernel segments, as these are its only data dependences on other nodes. Because the scout does no real work (and assuming that broadcasts are infrequent, which experience indicates is the usual case) the scout is usually executing at a point in the control path far ahead of the other nodes.

The scout node tracks all data placement (where each kernel variable is stored) and also identifies all data transfers required to prefetch remote IN data to a node executing a kernel segment. When a data prefetch is required, the scout sends a message to the receiving node (in our current protocol) saying, for example, "at segment 1206, you will need the value of X computed by node 85 at segment 724". The receiving node allocates storage for X, posts the receive, and then requests the value from node 85, which sends the value as soon as it finishes executing segment 724. Ideally, the scout message arrives before the sending node reaches segment 724 and the prefetch is complete before the receiving node reaches segment 1206.

One of the advantages of scouting as a basis for prefetching of array slices is that the computation of array subscript values is ordinarily part of the control computation, a part that is replicated on the scout node. Thus the scout knows exactly which parts of an array need to be prefetched.

Software Prototypes

The conceptual model sketched above is language and architecture independent, but different languages and architectures raise different engineering issues as to feasability and performance. We have explored the engineering issues in a series of software prototypes that target large Fortran 77 codes for execution on the Intel iPSC/860 (work done at ICASE and the University of Virginia). Software prototypes have been running since June, 1991 on the 32-node Intel at ICASE. We are currently running the 5th prototype, with version 6 under development. Test codes used to date have included several of the Perfect Club benchmarks, some of the NAS benchmarks, and a 4500 line NASA Langley CFD code.

All the techniques discussed in this paper are implemented in the current software (Version 5), except for dynamic storage allocation for multidef variables. Two restrictions apply: (1) memory requirements for the source code must be within single node memory limits (8 MB on the Intel iPSC/860), and (2) annotations must be used to identify kernel variables, mark kernel segments, and identify IN and OUT variables of these segments (Version 5 does not parse the Fortran source code). Version 6 will be the first "full-up" prototype, now targeted for 4th quarter, 1992. Version 6 will remove both the limitations noted above and will represent our first attempt to provide a fully automatic solution to the dusty deck problem.

Comparison with Data Parallel Methods

The following questions and answers attempt to draw out in more detail some of the key technical differences between the kernel-control parallel (KCP) and data parallel (DP) approaches. Because there are a variety of data parallel approaches, we try to provide a reasonably accurate assessment of what is being proposed for production software rather than for research systems.

Similarities. Both KCP and DP methods generate a single parallel program (load file) from the original source text, using traditional compiler analysis methods to insert communication calls into the generated code at appropriate points. Both methods use the SPMD model of execution, with one process per node. True data dependences are resolved by using message passing to send a copy of the data from the node "owning" it to the node needing a local copy.

Q: Grain size. What is the grain size of computation and communication?

Data parallel: Loop iterations are the basic computational grain-size; array elements or slices are the usual communication grain-size.

Kernel-control parallel: Whole loops, loop nests, conditionals, I/O statements, straight-line sequences, subprogram bodies, or loop iterations are all used as kernel segments (computational grains) as decided by the compiler. The communication grain size depends on the size of IN data sets for kernel segments. Each contiguous block of IN values for a segment is sent as a separate message, so a message may contain a whole array or COMMON block, an array slice, or a single element.

Q: Scheduling and load balancing. When are decisions about data placement and computation placement made?

Data parallel: Largely statically, during compilation. The compiler determines the distribution, alignment, and placement of array elements at each point in the source code, using information from user-provided annotations. Some DP versions permit redistribution of data at specified points during execution. Computation is scheduled statically based on data placement, using the "owner computes" or other scheduling heuristics.

Kernel-control parallel: Largely dynamically, during execution. Files and associated I/O operations are assigned statically to nodes. All other computation is scheduled dynamically, using heuristics that attempt to balance the load and minimize communication. Data values are left on the node where they are computed. Computation moves to the node with the IN data when appropriate or moves to a lightly loaded node. Each execution of a kernel segment is scheduled anew, taking account of the current placement of data and the computational load.

Q: Large array model. How are large arrays broken up, distributed, and accessed?

Data parallel: The programmer decides on a regular pattern of large array decomposition and alignment, using annotations to express the pattern. This pattern is mapped to the processor set statically, based on annotations. Regular access patterns in loops should allow local data to be used for most fetches; irregular access patterns force remote fetches and are costly. Arrays cannot generally be viewed as linear sequences of elements that can be reshaped arbitrarily.

Kernel-control parallel: The standard Fortran 77 array model is used. Arrays may be viewed as linear sequences of elements, and reshaping is allowed (full COMMON, EQUIVALENCE, and file I/O semantics). Array decomposition is determined dynamically during execution. Array slices are left on the nodes where computed, and local copies of slices are fetched to other nodes when needed as IN values for kernel segments. The programmer is not involved in placement decisions, but use of "contiguous slice" referencing patterns will improve performance (similar to vectorization).

Q: Prefetching. How effectively can prefetching of data be optimized? Is data ever prefetched and then not used? Is data ever prefetched again when a local copy is still available and fresh?

Data parallel: Prefetching is largely controlled by the array decomposition and alignment directives, which attempt to move data to the right local memory before it is referenced. This prefetching is "blind" in that it is not based on accurate knowledge of actual upcoming referencing patterns. Array elements may be moved to a node and never used. Values may be refetched repeatedly even though a local copy is still fresh.

Kernel-control parallel: Prefetching is based on the IN data sets identified for a kernel segment. Prefetching based on scout messages is theoretically optimal — prefetching will begin as soon as the data has been computed. In practice, the scout speed and IN data set size will determine actual effectiveness. If a kernel segment actually uses each IN value, then scout-based prefetching is entirely accurate. Only data that is used is prefetched and data is never prefetched again if the receiver still has a fresh copy.

Q: Performance tradeoffs. Which run-time elements are likely to have the most overhead?

Data parallel: Our best guess is: (1) synchronization for access to shared data, (2) sub-optimal "blind" prefetching, (3) load imbalance caused by static scheduling of loop iterations of varying cost.

Kernel-control parallel: Our best guess is: (1) dynamic scheduling of kernel segments, (2) slowdowns in prefetching when the scout waits for a broadcast value, (3) storage allocation and housekeeping for multidef variables, and (4) load imbalance caused by poor dynamic scheduling and irregular communication patterns.

Conclusions

Both data parallel and kernel-control parallel methods are unproven for large codes on MIMD distributed memory machines. Despite some superficial similarities, the two approaches differ radically in their choice of performance tradeoffs in the execution model. Experience with KCP prototypes indicates that it is difficult to predict the effective performance of these various run-time choices without extensive experimentation with prototypes. The KCP run-time advantage in optimal prefetching and the lack of synchronization looks appealing, but it may be that the additional overheads incurred in other parts of the model will outweigh these gains. We look forward to performance comparisons between full prototypes of both models in the near future.

To some audiences, the use of existing language models in the KCP approach is a major advantage because it allows dusty decks to be ported to parallel machines without change. To others, the use of new language models in the DP approach is an advantage because it forces restructuring of applications algorithms and data representations into forms more amenable to MIMD parallel execution. New applications codes are the most likely candidates for the DP approach; old codes are the obvious target for the KCP approach, and each represents a sizable user community.

[1] G. Steele, "High Performance Fortran: Status Report," (this workshop).

[2] B. Chapman, P. Mehrotra, and H. Zima, "Programming in Vienna Fortran," *Scientific Programming*, 1, 1, 1992, 31-50.

[3] G. Fox, et al, "Fortran D Language Specification," TR90079, CS Dept., Rice University, March, 1991.

[4] T. Pratt, "Program Analysis and Optimization through Kernel-Control Decomposition," *Acta Informatica*, 9 (1978), 195-216.