Automatic Parallelization of Recursive Procedures

Manish Gupta
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598
mgupta@us.ibm.com

Sayak Mukhopadhyay
Mobius Management Systems
Rye, NY 10580
smukhopa@mobius.com

Navin Sinha
IBM Global Services India
Bangalore 560017, India
r1navink@in.ibm.com

Abstract

Parallelizing compilers have traditionally focussed mainly on parallelizing loops. This paper presents a new framework for automatically parallelizing recursive procedures that typically appear in divide-and-conquer algorithms. We present compile-time analysis to detect the independence of multiple recursive calls in a procedure. This allows exploitation of a scalable form of nested parallelism, where each parallel task can further spawn off parallel work in subsequent recursive calls. We describe a run-time system which efficiently supports this kind of nested parallelism without unnecessarily blocking tasks. We have implemented this framework in a parallelizing compiler, which is able to automatically parallelize programs like quicksort and mergesort, written in C. For cases where even the advanced symbolic analysis and array section analysis we describe are not able to prove the independence of procedure calls, we propose novel techniques for speculative run-time parallelization, which are more efficient and powerful in this context than analogous techniques proposed previously for speculatively parallelizing loops. Our experimental results on an IBM G30 SMP machine show good speedups obtained by following our approach.

1 Introduction

While parallel machines are increasingly becoming ubiquitous, developing parallel programs continues to be a tedious and error-prone task. Parallelizing compilers offer the potential advantage of exploiting parallelism in sequential programs, which are easier to write, maintain, and port to other machines. However, current parallelizing compilers are usually able to detect only loop-level parallelism, and hence cater only to a restricted class of programs. This paper presents an approach to parallelizing recursive procedures, which enables automatic parallelization of a class of applications that cannot be parallelized by the previous generation of compilers.

Our work exploits a scalable form of nested functional parallelism by detecting the mutual independence of multiple recursive calls inside a procedure. This approach is particularly well-suited to parallelizing applications employing a divide-and-conquer algorithm. Such an algorithm usually consists of three steps: (i) *divide* – the problem is partitioned into two or more subproblems, (ii) *conquer* – the subproblems are solved by recursive calls to the original procedure, and (iii) *combine* – the solutions to the subproblems are combined to obtain the final solution to the overall problem. Typically, the partitioning of the problem during the first step is performed such that the recursive calls during the second step can be executed independently.

The divide-and-conquer style of writing array based computations is likely to gain popularity in the future, as it allows better exploitation of the memory hierarchy in modern machines, particularly when used together with a recursively blocked array data layout [11]. Researchers have recently demonstrated such algorithms for linear algebraic computations, like matrix multiplication and QR factorization, that perform significantly better than equivalent level-3 BLAS (basic linear algebra subprogram) codes [5, 11]. While our compiler does not yet adequately analyze arrays using blocked layout, this work provides a foundation for parallelizing these codes as well in the future.

This paper makes the following contributions:

- It presents a compile-time framework, using powerful symbolic array section analysis interprocedurally, to detect the independence of multiple recursive calls in a procedure. We describe design decisions that make our analysis efficient, yet effective.
- It presents novel techniques for speculative run-time parallelization to deal with cases that defy compile-time analysis. These techniques are more efficient and powerful, in the context of parallelizing divide-and-conquer computations, than analogous techniques proposed earlier for speculatively parallelizing loops [23].
- It presents experimental results, demonstrating the capability of our compiler in automatically parallelizing

programs like quicksort and mergesort (written in C) and showing the impact of the run-time system design on performance.

The rest of this paper is organized as follows. Section 2 discusses related work in this area. Section 3 presents a motivating example for our work. Section 4 describes the compile time analysis to infer the independence of various recursive calls in a procedure. Section 5 describes a run-time system that efficiently supports the parallelization of recursive procedures. Section 6 discusses techniques for speculative parallelization that can be used to effectively deal with cases where compile-time analysis alone is not accurate enough to detect parallelism. Section 7 presents the experimental results obtained on an IBM G30 shared memory multiprocessor. Finally, Section 8 presents conclusions and ideas for future work.

2 Related Work

A great deal of previous research on using interprocedural analysis in parallelizing compilers [12] has been directed towards parallelizing loops containing procedure calls, and does not attempt to find non-loop parallelism.

Girkar and Polychronopoulos have proposed a framework for exploiting task-level parallelism in programs [7]. They use a *hierarchical task graph* to represent control and data dependences between task units such as loops, basic blocks or procedures, in a hierarchical manner. Ramaswamy et al. have presented an approach that uses compile-time analysis to combine task-parallelism with data-parallelism [22]. These researchers do not discuss handling recursive procedures.

In the context of functional languages like Scheme, Harrison has proposed *recursion splitting* [16], a transformation which converts a recursive procedure into loops. These loops are parallelized by standard loop parallelization methods and a special technique called *exit-loop parallelization*, which handles recurrences in control flow cycles with exits. In contrast, our work directly recognizes the parallelism between different recursive procedure calls, without attempting to transform them to loops.

Several researchers have used pointer analysis to recognize parallelism in programs that use linked data structures like trees and linked lists [14, 19]. These techniques are complementary to our work, which relies on more detailed array section analysis to work with array-based computations (including those accessing arrays via pointers). Several other systems, such as Cilk [2] and the Multilisp language [17] provide run-time support to exploit parallelism between recursive calls.

Rinard and Diniz present commutativity analysis [24] for parallelizing irregular, object-based computations, including recursive procedures, which perform operations that commute. Their work does not handle methods which write into their reference parameters.

```
QuickSort(A, low, high) {
    if low < high
        p = A[low]
        lastsmall = low
        for i = low + 1 to high do
            if (A[i] < p) then
                lastsmall = lastsmall + 1
                Swap(A[lastsmall], A[i])
            end if
        end do
        Swap(A[low], A[lastsmall])
        QuickSort(A, low, lastsmall)
        QuickSort(A, lastsmall + 1, high)
    end if
}
Swap(x, y) {
    t = x;    x = y;    y = t
}
```

Figure 1. Program example: quicksort

Rugina and Rinard have independently developed techniques to automatically parallelize divide-and-conquer applications [25]. They achieve roughly similar results as our work. However, the two approaches differ in the following ways. First, our work uses different symbolic analysis techniques that extend those employed in production compilers: we use novel extensions to (and a combination of) guarded array region analysis [8], symbolic range propagation [1], and generalizations of induction variable analysis [6], while Rugina and Rinard use abstract interpretation techniques and correlation analysis of variables, to obtain precise enough array access information to support detection of parallelism. We believe that their analysis [25] would fail to detect parallelism in our motivating example, described in Section 3. Second, our work is unique in presenting techniques for speculative parallelization of recursive procedures, which can be used to parallelize programs where compile-time analysis alone is incapable of extracting parallelism.

Moon and Hall use predicated array data flow analysis for automatic parallelization [20]. They derive efficient run-time parallelization tests from these predicates.

3 Motivating Example

Consider a quicksort program, taken from [18], shown in Figure 1. The *QuickSort* procedure selects an element $p = A[low]$ as a pivot element around which $A[low : high]$

is partitioned. After partitioning the array such that elements smaller than the pivot element are moved before the position *lastsmall*, it makes two recursive calls to sort the two parts of the array A. These recursive calls operate on distinct sections of the array. The compiler requires advanced symbolic analysis and interprocedural array section analysis to infer that there is no overlap between data referenced in the two calls. Hence, these calls can be executed in parallel. The parallelism between the two procedure calls can be exploited at each successive level of recursion. Hence, the run-time system can allow the number of parallel work items to be steadily increased until it is no longer profitable to create additional parallel work.

4 Compile-Time Analysis

The basic goal of our compile-time analysis is to determine the independence of different recursive calls in a procedure, based on the absence of data and control dependences between them. We have added these analyses to an existing optimizer, the Toronto Portable Optimizer, which forms part of both Fortran 90 and C compilers. It works on a common intermediate language, W-Code, that is produced by different front-ends. These analyses are performed after an interprocedural application of classical optimizations like constant propagation, copy propagation, and dead code elimination.

4.1 Basic Array Reference Processing

The optimizer deals with different kinds of arrays, statically or dynamically allocated (possibly accessed via pointers), and those originating from different language constructs (and hence, with different data layout orders), in a unified manner. It sees array references in the form of $*(base + disp)[sub]$, where $(base + disp)$ is the address where the 0^{th} array element would be located (it is an adjusted, hypothetical, address if zero is not a valid subscript), and *sub* is the linearized subscript. For an explictly named array (say, $A[]$), the *base* expression is an address of the variable (A), while for an array accessed through a pointer, it represents the value of the pointer. The compiler performs flow-sensitive, context-sensitive, interprocedural pointer analysis [3], which is used to provide aliasing information to array section analysis. If an array is aliased with another variable, an access to that aliased variable is regarded as potentially touching any part of the given array.

The subscript information for a multidimensional array, which appears to the optimizer in an almost linearized form (the information about individual dimensions is retained via separate indexing opcodes for different array dimensions, generated by the Fortran 90 and C front-ends), is processed to recover separate subscript values for distinct strides being used in the (linearized) subscript expression. Currently, our techniques fail to accurately handle cases where the base address is not invariant, but can be extended to cover cases where direct additions to the base address (rather than subscripts) are used to access different array elements.

4.2 Interprocedural Array Section Analysis

For each procedure, the compiler obtains the following information about formal parameters and global variables:
MAYDEF: variables that may be written.
MUSTDEF: variables that are definitely written.
UPEXPUSE: variables that may have an *upward exposed use*, i.e., a use not preceded by a definition along a path from the procedure entry.

For arrays, this information is represented using a list of *Guarded Array Regions (GARs)* [8]. A GAR is a tuple $\langle G, D \rangle$, where D is a bounded *Regular Section Descriptor (RSD)* [13] for the accessed array section, and G is a guard that specifies the condition under which D is accessed [8]. The bounded RSD provides information about the lower bound, upper bound, and the stride in each dimension, and thus represents sections that may be described using Fortran 90 triplet notation. Each bound, stride or predicate expression may include constants or scalars that are formal parameters or global variables, or it may take a statically unknown value, represented as $\perp$.

While performing a union or subtraction operation on GARs, if the result cannot be accurately represented using a single GAR, it is represented as a list of GARs. In our current implementation, we also extend the representation of a GAR in a similar manner: it consists of a list of RSDs associated with a single guard. For the sake of efficiency, the number of entries in each list is not allowed to exceed a parameterized constant value (our current implementation uses a value of 5). Whenever an operation would lead to the size of the list exceeding this limit, entries from the list are heuristically chosen for combining. Furthermore, each time the array access is summarized for an *interval*, i.e., a loop or the entire procedure, the list is compressed if some entries in the list can be combined without losing accuracy.

The procedure for computing MAYDEF, MUSTDEF and UPEXPUSE is described in [8]. Our adaptation of this procedure, apart from applying it to handle recursion as well, differs from [8] mainly in the following ways: (i) we use more sophisticated symbolic analysis, discussed in Section 4.5, to obtain more precise array section information, and (ii) we control the cost of symbolic analysis by performing symbolic expression simplification only when compressing the list of GARs, rather than each time a set operation is performed on two GARs.

For the purpose of illustrating our analysis to detect parallelism between recursive calls, we outline the procedure

briefly for computing UPEXPUSE, which is done in a backward pass over the control flow graph (CFG). It is based on computing the sets UEUIN/UEUOUT, at entry/exit of each CFG node, which keep a cumulative account of upward exposed uses starting backwards from the exit node. UEUOUT_i is initialized to $\emptyset$ for the exit node, and the value of UEUIN obtained for the entry node is used as the summary UPEXPUSE information for the procedure. The following basic equations describe the computation for a node i, which could represent a basic block, loop, or procedure.

$$\text{UEUIN}_i = [\text{UEUOUT}_i - \text{MUSTDEF}_i] \cup \text{UPEXPUSE}_i \quad (1)(2)$$

$$\text{UEUOUT}_i = \cup_{s \in succ(i)} \text{UEUIN}_s \quad (3)$$

While applying Equation 3 above, the condition from the predicate associated with the control flow edge from node i to node s is added to the guard of the GAR representing UEUIN_s. We use interval analysis to handle loops [8, 10].

The overall interprocedural analysis is performed in a reverse topological sort order over the call graph. In the absence of recursion, this ensures that while encountering a call to procedure p, the summary MAYDEF, MUSTDEF, and UPEXPUSE information recorded for p is already available at its call site. This summary information is translated to replace formal parameters by actual arguments used at the call site. Our current implementation performs this translation accurately only for cases where the array shape does not change across the procedure call. The aliasing information, obtained separately by the compiler, is taken into account by modifying the array section information conservatively, as mentioned in Section 4.1.

Some observations We have found information from guards to be crucial for accurate array section analysis. For example, while computing UPEXPUSE for A in the *QuickSort* procedure shown in Figure 1, consider the apparently simple union operation of $A[low : low : 1]$, from the read access before the loop, with $A[low+1 : high : 1]$, from the read access in the loop. The compiler would fail to obtain the resulting section, which may be over-approximated for correctness, as $A[low : high : 1]$, without information from the predicate $(low < high)$ that guards the procedure body. Clearly, $A[low : high : 1]$ would not correctly represent $A[low : low : 1] \cup A[low+1 : high : 1]$ if $(low > high)$, since it would be an empty section under that condition, and hence would incorrectly exclude the element $A[low]$.

Using a *list* of GARs gives us the flexibility of attaching and exploiting guard information in a natural, bottom-up manner, rather than having to propagate guard information eagerly inside program intervals. In the above example, the entries for array sections $A[low : low : 1]$ and $A[low+1 : high : 1]$ would be kept separate initially. The predicate $(low < high)$ is added to the list of GARs just before UPEXPUSE is summarized for the *QuickSort* procedure. It is during the compression of the list of GARs attempted while summarizing UPEXPUSE for the procedure, that the two array sections are combined to obtain $A[low : high : 1]$.

4.3 Handling of Recursion

We use fixed point iteration to deal with recursion. For example, let UPEXPUSE_p^i denote the upward exposed use computed at the end of analysis pass i through the procedure p. During pass i, UPEXPUSE_p^{i-1} is used to compute the upward exposed use summary for each recursive call (UPEXPUSE_p^0 is initialized to $\emptyset$). The iterative procedure is repeated until the solution to UPEXPUSE_p converges.

For the sake of efficiency, we bound the number of iterations to two. If convergence is not reached by the end of the second pass, UPEXPUSE_p is set to $\perp$, which denotes an unknown value. We expect two passes to be sufficient for the data flow solution to converge for a typical divide-and-conquer application. While the recursive invocations to the procedure solve different parts of the problem, either the *partition* step which precedes it or the *combine* step which follows it, or both, typically access the complete data. As a result, the summary data access information from the recursive procedure call sites, corresponding to the subproblems being solved, do not contribute anything in addition to the data access descriptor obtained for the overall procedure at the end of the first analysis pass.

4.4 Check for Independence between Call Sites and Code Motion

Different recursive calls can be executed in parallel if there are no control or data dependences among them. Currently, we only consider recursive calls in the same basic block in the procedure, which trivially implies that there is no control dependence between them. Consider two recursive call sites P_1 and P_2, where P_1 appears before P_2 in a given basic block. If there are more than two recursive calls, they can be handled by a trivial extension of the analysis we shall now describe.

P_1 and P_2 can be executed in parallel only if there is no chain of data dependences, possibly via other statements in the basic block, from P_1 to P_2. This is tested by constructing a data dependence graph with nodes corresponding to statements in that basic block from P_1 to P_2, and verifying that there is no path along data dependence edges from P_1 to P_2. We check for, respectively, true, anti, and output dependences, from statement S_1 to statement S_2, with the following tests: $\text{MAYDEF}_{S_1} \cap \text{UPEXPUSE}_{S_2} \neq \emptyset$, $\text{UPEXPUSE}_{S_1} \cap \text{MAYDEF}_{S_2} \neq \emptyset$, and $\text{MAYDEF}_{S_1} \cap$

$\text{MAYDEF}_{S_2} \neq \emptyset$. If parallelization is prevented by only anti and output dependences, those false dependences can be overcome by introducing extra storage and renaming variables.

Once the complete independence of P_1 and P_2 is established following the above procedure, all statements between P_1 and P_2 are partitioned into three, possibly empty, sets: (i) statements to which there is a direct or indirect data dependence from P_1, (ii) statements from which there is a direct or indirect data dependence to P_2, and (iii) all other statements. The first set of statements are moved after P_2 in the same basic block, the second set of statements are moved before P_1, and statements belonging to the third set may be moved either before P_1 or after P_2. Within each set, the ordering amongst statements is preserved under this code motion. This results in all recursive calls being placed adjacent to each other, thus isolating the task being parallelized from the rest of the computation.

4.5 Symbolic Analysis

We now describe some of our symbolic analysis techniques that enable the detection of parallelism to be done more effectively. Essentially, we require an accurate estimation of the *range* of values taken by an expression appearing in the descriptor representing MAYDEF or UPEXPUSE information for an array.

Overview of Symbolic Range Computation Framework Our framework has some similarities to Blume and Eigenmann's demand-driven *symbolic range propagation* algorithm [1], and Tu and Padua's work on gated SSA-based symbolic analysis [26]. Unlike their work, we apply this analysis only while *widening* a list of GARs, for summarizing it with respect to a program interval. Using symbolic analysis at that stage also helps compress the list of GARs. In our framework, the predicate expression appearing in the GAR can directly be used to refine the range of expressions corresponding to data access, so that we do not have to follow gated-SSA links, as in [26], or follow dominating control flow edges, as in [1]. The expression, whose range is to be computed, is first expanded by substituting each variable having a unique reaching definition with the `rhs` expression used in its assignment. It is then subjected to algebraic simplification, which uses extensive constant folding, exploits algebraic properties like commutativity, associativity, and distributivity of the relevant operations over integers, and uses boolean equalities like de Morgan's laws for simplifying predicate expressions. The bounds on the value of any variable appearing in the resulting expression are obtained, as explained below, by (i) substituting the known bounds of that variable from the interval descriptor, if it is an induction variable or monotonic variable, and (ii) taking into account the predicate in the GAR. These bounds are, in turn, used to obtain symbolic range information for the expressions appearing in the GARs. We use the notation $e_{\min}$ to denote the minimum value an expression e can take, and $e_{\max}$ to denote its maximum value. We note that if no bounds information is available for a variable x, its range expression simply appears as $[x_{\min}, x_{\max}]$, which is interpreted for an interval L as $[x, x]$ if x is invariant with respect to L, and $[-\infty, +\infty]$ if x is not invariant with respect to L[1].

Monotonic Variables Our compiler recognizes *monotonic variables*, which are incremented or decremented conditionally, but monotonically, within a loop body [6]. We extend the conventional definition of monotonic variable with respect to a program interval [6] to include the interval in which it is initialized, if it is not used prior to its initialization, and if it satisfies the monotonicity property after its initialization. For example, in Figure 1, the variable *lastsmall* is regarded as monotonic with respect to the outermost interval in *QuickSort*, which represents the overall procedure. Our algorithm is otherwise similar to the one proposed by Gerlek et al. [6], and is applied as an extension to induction variable recognition. For each interval, it identifies monotonic variables and the maximum increment or decrement of their values in a single iteration of the interval. Therefore, for countable loops such as `do` loops, symbolic lower and upper bounds can be established on the value of a monotonic variable. For loops without a fixed count, such as general `while` loops, a symbolic lower (upper) bound can be obtained for a monotonically increasing (decreasing) variable. Our algorithm propagates information about the monotonicity of a variable to outer intervals that have no other definition of that variable, and also to the interval which initializes it, provided that initial definition *dominates* all uses of the variable in that interval.

Using Predicate in GAR to refine Symbolic Range of a Variable The predicate is first subjected to expression simplification and is transformed into a form that uses only conjunctive and disjunctive operations over terms representing relational expressions of the form $(expr_i \oplus expr_j)$, where $\oplus \in \{<, \leq, =, >, \geq\}$. Given a variable x whose range of values is to be computed, if a relational expression can be expressed as $(x \oplus expr)$, a constraining range is obtained for x. For example, $(x \leq expr)$ implies a range $[x_{\min}, expr_{\max}]$ for x (i.e., $x_{\max} = expr_{\max}$). Similarly $(x \geq expr)$ implies a range $[expr_{\min}, x_{\max}]$ for x (i.e., $x_{\min} = expr_{\min}$). If the relational expression cannot be

[1] We loosely use the term $+\infty$ to denote the highest possible value of a variable and $-\infty$ to denote the lowest possible value of the variable. Our implementation uses the information on data type of a variable to use the appropriate number or flag. For instance, for an unsigned integer variable, $-\infty$ corresponds to zero.

expressed in the form $(x \oplus expr)$, the range for x from that expression trivially remains $[x_{\min}, x_{\max}]$. Ranges are obtained from conjuction and disjunction of relational expressions r_1 and r_2 using the following two rules:

$$range(r_1 \vee r_2) = range(r_1) \cup range(r_2)$$
$$range(r_1 \wedge r_2) = range(r_1) \cap range(r_2)$$

While applying the above rules, any unresolved term of the form $x_{\min}$ is treated as $-\infty$ and of the form $x_{\max}$ is treated as $+\infty$. For example, consider a GAR, $G = \langle x \leq 100 \wedge x \leq y, A[x+1] \rangle$, being expanded with respect to a loop in which x is a monotonically increasing variable with an initial value of 0 (i.e., $x_{\min} = 0$) and y is a monotonically decreasing variable with an initial value of 50 (i.e., $y_{\max} = 50$). We simplify the GAR by establishing a range for x as:

$$[x_{\min}, 100] \cap [x_{\min}, y_{\max}] \equiv [0, 100] \cap [0, 50] \equiv [0, 50],$$

thus obtaining $G = \langle x \leq 100 \wedge x \leq y, A[1 : 51] \rangle$.

4.6 Program Example

Let us revisit the program shown in Figure 1 and illustrate some salient aspects of our analysis on that program. The symbolic analysis identifies the variable *lastsmall* as a monotonically increasing variable with respect to the `for` loop, with a maximum increment of +1 in each iteration of the loop. The interprocedural side-effects summary for the procedure calls outside the loop shows that *lastsmall* is not modified at any of those call sites. Hence, the symbolic analysis is able to propagate the bounds information for this variable as $low : high$ for the entire procedure interval.

The array section analysis, in the first pass through the *QuickSort* procedure, obtains both MAYDEF and UPEXPUSE information for the array A as $A[low : high]$, by regarding the MAYDEF and UPEXPUSE information at the recursive call sites as $\emptyset$ initially and by exploiting the bounds information for *lastsmall*. During the second pass through the procedure, this parameterized array section information is translated to $A[low, lastsmall]$ and $A[lastsmall + 1, high]$ respectively at the two recursive call sites. The union operation of each of these sections with $A[low : high]$ leads to no change, given the bounds information on *lastsmall*. Hence, the converged solution for both MAYDEF and UPEXPUSE for the procedure with respect to the array A is obtained as $A[low : high]$.

The two recursive calls appear in the same basic block, as required by our parallelization analysis. There is no intervening statement between them. A data dependence between the two call sites is ruled out by the tests described in Section 4.4, as the intersection of $A[low, lastsmall]$ with $A[lastsmall + 1, high]$ is identified as null. Hence, we recognize the two recursive calls to *QuickSort* as forming a parallel task.

5 Run-Time Support

Our run-time library support for parallel execution of recursive procedures is an extended version of that used for supporting dynamic scheduling of loops [15]. The compiler uses the *outlining* transformation to convert the given parallel loop or task into a subroutine, to be scheduled among threads using a single shared work-queue. A single master thread, which also executes the serial parts of the program, creates slave threads that will participate in parallel execution, at the beginning of the program. Each work item put on the queue represents the multiple recursive invocations made in the same procedure, which are to be executed in parallel. A single recursive call, referred to as a chunk of work item, is the unit of work allocated to a given thread. While the initial parallel work item is placed on the queue by the master thread, slave threads can subsequently enqueue work items as well, since we support nested parallelism.

Each thread which enqueues a parallel work item also participates in its execution. In fact, after completing its chunk, if there is no available work within that item, it looks for work on other items that may have been enqueued by other threads. When there is no more work available, it waits for the work item that it enqueued (on which other threads may be working) to be completed. Once that item, corresponding to all recursive calls made in the original procedure assigned to it, is completed, it resumes execution of the remaining code in the procedure after the recursive calls. This strategy allows all threads to participate in the execution of parallel tasks created at deeper levels of recursion before being blocked at higher levels, waiting for children recursive calls to be over.

As the computation proceeds, the parallel work items enqueued at deeper levels of recursion increasingly represent smaller grains of computation. Hence, we need a mechanism to control the creation of parallel work items. We use a heuristic approach which bounds the number of parallel units of work to a constant multiple of the number of processors executing the program. The value of this constant, that we refer to as the *parallel load factor*, can be tuned based on the need for load balancing. If this parameter is set to one, the number of parallel tasks would be equal to the number of processors. Although the total overhead of the enqueue and the dequeue operations on the task queue would be minimized in this case, it would leave little room for load-balancing if different tasks require different amounts of time. By increasing the total number of parallel tasks to a higher value, we can obtain better load balance at the expense of increased scheduling overhead.

The actual mechanism used in the run-time system is to keep track of the depth of recursion and limit the depth beyond which further recursive calls are executed serially. Let

```
QuickSort(A, l, h) {
    if l < h
        q = Partition(A, l, h)
        QuickSort(A, l, q)
        QuickSort(A, q + 1, h)
    end if
}
Partition(A, l, h) {
    x = A[l];    i = l − 1;    j = h + 1
    while TRUE do
        repeat j = j − 1
        until A[j] ≤ x
        repeat i = i + 1
        until A[i] ≥ x
        if i < j
            Swap(A[i], A[j])
        else
            return j
        end if
    end do
}
```

Figure 2. An alternate form of pseudocode for quicksort

k be the fixed number of recursive calls to the procedure which are mutually independent. If d is the number of levels of recursion until which parallel work items are created, the total number of parallel chunks of work created is k^d. If P is the number of processors and the parallel load factor is c, then d is obtained as $\lceil \log_k(c * P) \rceil$. The run-time parallel task setup routine, which is called from the outlined procedure that represents the parallel task, keeps track of the depth of recursion, and either enqueues a new work item on the queue or executes the original calls serially.

6 Speculative Parallelization

It may not always be possible for the compiler to establish the absence of data dependence between different recursive calls. Consider a slightly different form of `QuickSort` program [4], shown in Figure 2. For array accesses in the `while` loop of procedure `Partition`, the compiler is able to establish the bounds on j as $[j_{\min}, h]$, and on i as $[l, i_{\max}]$. The write accesses to $A[i]$ and $A[j]$ are guarded by the conditional $i < j$, which when combined with the bounds information for i and j, can allow the compiler to obtain the MAYDEF section of A as $A[l : h]$. However, the compiler is not able to obtain tight bounds on the section of A with read access, so that UPEXPUSE for A is obtained as $A[-\infty : \infty]$. Therefore, the two recursive calls to `QuickSort` appear to have data dependence between them.

In order to deal with the problem of imprecise array section analysis preventing parallelization, we propose a novel framework for run-time parallelization. Based on symbolic analysis, the compiler speculatively executes the multiple recursive calls in parallel and performs efficient run-time tests to verify the legality of parallelization. Our framework is inspired by that proposed by Rauchwerger [23] for parallelizing `do` loops, but incorporates many improvements.

Given multiple recursive calls of a procedure being speculatively parallelized, the calls other than the first recursive call are regarded as speculative. These calls are made to a modified version of the procedure, in which all writeable variables are allocated from the private memory of the thread. Before beginning parallel execution, the upward exposed use sections of those variables are copied in from shared memory. After completion, if the run-time test, described below, indicates successful parallelization, live data from each of those procedures is copied back into shared memory in the sequential order of their calls, thus committing the results of the parallel computation. If the run-time test shows a data dependence between the recursive calls, then data from the speculative calls is discarded and they are re-executed sequentially. This approach significantly reduces the penalty of mis-speculation if the procedure calls turn out to have dependences, since the computation from the first recursive call (which is executed in parallel with other recursive calls) is not thrown away, unlike the framework proposed in [23], where the entire loop computation is discarded if the loop is found to be serial.

The compiler decides to use speculative parallelization only when it fails to obtain symbolic bound(s) for MAYDEF or UPEXPUSE section of one or more arrays, and the conservative setting of that bound to $-\infty$ or $+\infty$ leads to an apparent data dependence from one recursive call site to another. For each such array, an additional *bounds array* is created, which records the lowest and the highest subscript positions, in each dimension, for read and write accesses in the procedure, if the conservative values obtained for those bounds at compile-time are $-\infty$ or $+\infty$. By taking advantage of the information about monotonicity of subscript expressions in a loop, based on the recognition of induction and monotonic variables, it may be possible to hoist much of the computation to update the bounds array outside the loop. Previously proposed run-time parallelization schemes, such as [23, 9], use shadow arrays to record read and write access to each array element. Although those schemes can detect parallelism in more cases for arbitrary data access patterns, they incur greater overhead of check computation, which seems unnecessary for applications where potentially parallel computation partitions op-

Program	*Serial Time (sec)*			
	1M	*2M*	*5M*	*10M*
MergeSort	12.39	26.13	70.23	147.11
QuickSort1 Data Set 1	8.51	21.53	85.09	268.43
QuickSort1 Data Set 2	8.58	21.40	84.09	267.50
QuickSort2	6.78	14.66	37.90	N/A

Table 1. Serial execution times of programs

erate on contiguous data. Patel and Rauchwerger have independently proposed a similar scheme, using shadow interval structures, for loops [21].

Finally, at the end of speculative execution of the procedure calls, a run-time test is performed to check if the parallelization was valid. This test verifies the absence of true data dependence between recursive calls by checking that MAYDEF for the earlier recursive call has no overlap with UPEXPUSE for the later call, using information from the bounds array(s). Unlike the general LRPD test proposed in [23], in which anti and output dependences are also checked, our run-time test can safely ignore such storage-related dependences. The very mechanism of using private storage for each variable being written in the procedure (to support rollback of speculative computation) allows these storage-related dependences to be broken [9].

7 Experimental Results

We now present experimental results obtained on an IBM G30 shared memory machine with four PowerPC 604 processors. We have implemented the compiler analysis, code generation, and the run-time system described in this paper to support nested parallelization of recursive procedures. In the future, we also plan to implement our framework for speculative parallelization.

We selected three programs, written in C, which could not be parallelized by any other compiler at the time of our experiments. The first program is `Mergesort`, which applies the standard mergesort algorithm [4] on an array. The remaining programs, which we refer to as `QuickSort1` and `QuickSort2`, are based on the two versions of `QuickSort`, shown in Figures 1 and 2. `Mergesort` and `QuickSort1` were parallelized automatically by our compiler. For `QuickSort2`, where compile-time analysis alone cannot possibly succeed, we manually applied the techniques for speculative parallelization. In each case, we initialized the array to be sorted using pseudo-random data.

Figure 3 shows the speedups obtained on these programs for different array sizes, ranging from 1M (one million) to 10M elements. Table 1 shows the serial execution time for each case. We observed significant variation in the performance of the parallel version of `QuickSort1` for different data sets (due to different work partitions generated for different data sets), even though each data set was pseudo-randomly generated. Hence we have shown its performance for two separate data sets.

The speedups exhibited by `MergeSort` are quite consistent across different data sets, and vary between 2.8 and 3.0 on four processors. In `QuickSort1`, the distribution of load between different parallel tasks is highly data dependent, and hence it shows more variation across different data sets and data sizes. The speedups range from 2.2 to 3.0 on four processors. We observed lower speedups for `QuickSort2`. This can be attributed mainly to the extra overhead of copy-in and copy-out needed for the privatized data to support speculative execution. A different problem with a higher ratio of computation to data movement could benefit more from our speculative parallelization techniques.

Figure 4 shows the effect of parallel load factor (PLF) on the performance of programs for a fixed data size of 2M elements. In `MergeSort`, the partitioning of load between the parallel chunks of work is quite even. Hence, when using two or four processors (any number which is a power of two), the performance is better at lower PLF values due to the lower scheduling overhead. With three processors, there is a greater need for load balancing, as the parallel tasks do not fold evenly on to the available processors. Hence, a higher PLF leads to improved performance in that case. Similarly, the performance of `QuickSort1` improves as PLF is increased from 1 to 8. On every program, raising PLF from 8 to 16 hurts performance as the additional scheduling overhead has a bigger impact than improved load balance, which shows diminishing returns. On `QuickSort2`, a higher PLF leads to larger memory requirements and copy-in and copy-out overheads. Hence, the performance on one processor shows a clear degradation as PLF is increased. This effect is also seen as the number of processors is raised to two. However, as the number of processors is increased further, a higher PLF upto 8 improves performance as the need for load-balancing increases.

Finally, we note that our results could only be presented for a shared memory machine with four processors that we had access to. Given that our methods detect fairly coarse-grain parallelism, we expect significantly better speedups on parallel machines with larger number of processors.

8 Conclusions

Even the most advanced parallelizing compilers today are not able to detect the available coarse-grain parallelism in a large number of programs. In this paper, we have presented our efforts towards extending the scope of automatic parallelization to procedures with multiple recursive calls that operate on distinct data. Hence, our approach is par-

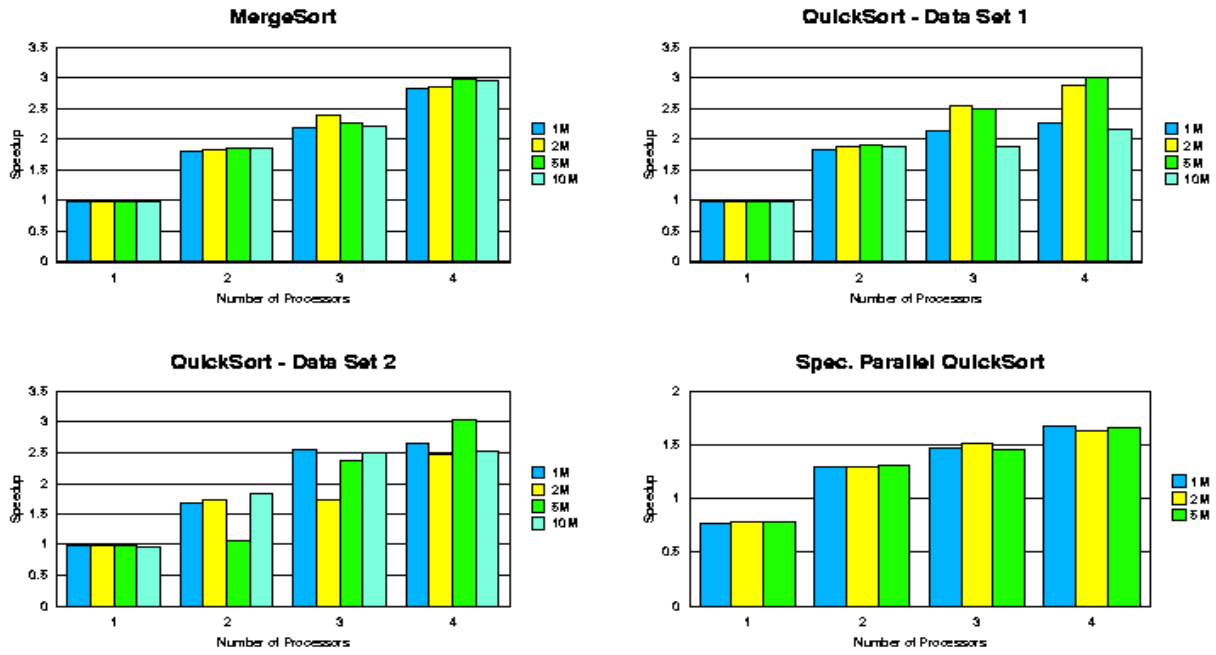

Figure 3. Performance results for different data sizes on IBM G30

ticularly well-suited to parallelizing computations that use divide-and-conquer algorithms. We have presented a sophisticated, yet practical, compile-time framework to detect and exploit this form of parallelism. We have also proposed novel run-time techniques to allow parallelization of programs for which compile-time analysis alone is not sufficient. In the future, we plan to extend our techniques to parallelize computations employing a recursively blocked array layout and to parallelize computations with apparent control dependences between procedures, possibly using a combination of compile-time analysis and run-time parallelization.

Acknowledgements

We wish to thank the members of the IBM Toronto Portable Optimizer team for their contributions to the compiler infrastructure used in this work. We would also like to thank the referees for their valuable comments.

References

[1] W. Blume and R. Eigenmann. Demand-driven, symbolic range propagation. In *Proc. 8th Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[2] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Santa Barbara, CA, July 1995.

[3] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 232–245, January 1993.

[4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1989.

[5] E. Elmroth and F. Gustavson. New serial and parallel recursive QR factorization algorithms for SMP systems. In *Proc. PARA'98 Workshop on Applied Parallel Computing in Large Scale Scientific and Industrial Problems*, Umea, Sweden, June 1998.

[6] M. Gerlek, E. Stoltz, and M. Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Transactions on Programming Languages and Systems*, 17(1):85–122, January 1995.

[7] M. Girkar and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), March 1992.

[8] J. Gu, Z. Li, and G. Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Proc. Supercomputing '95*, San Diego, CA, December 1995.

[9] M. Gupta and R. Nim. Techniques for speculative run-time parallelization of loops. In *Proc. SC '98*, Orlando, Florida, November 1998.

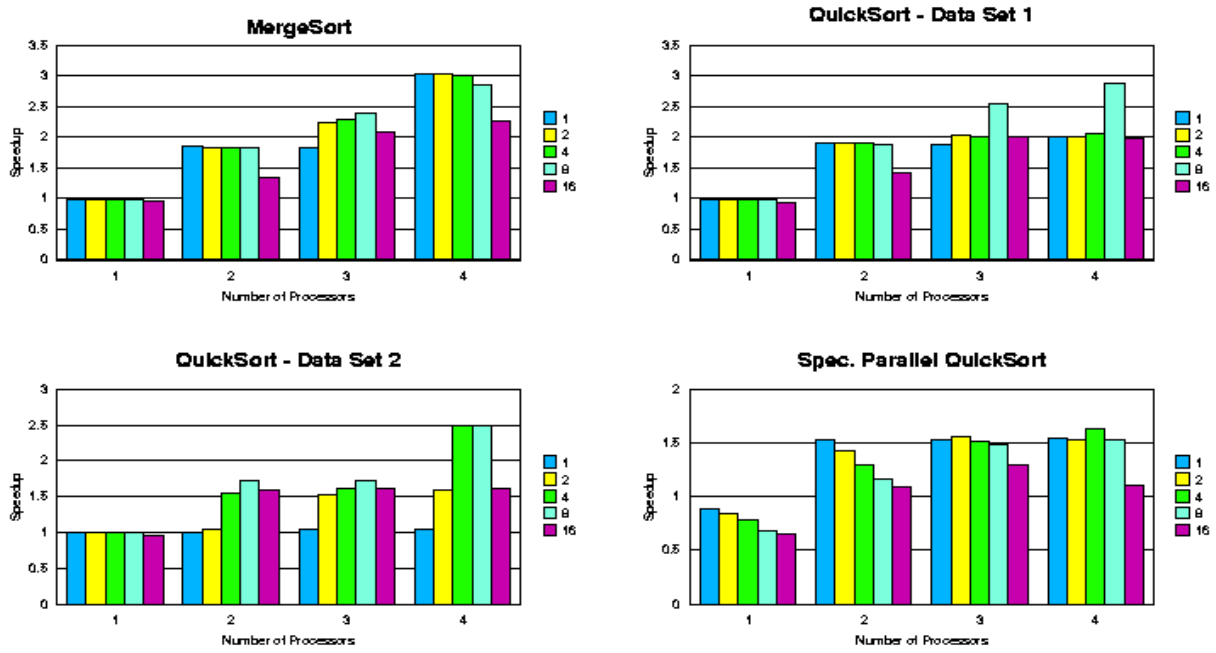

Figure 4. Performance results for different parallel load factors

[10] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(7), July 1996.

[11] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra systems. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[12] M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S.-W. Liao, and M.S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proc. Supercomputing '95*, San Diego, CA, December 1995.

[13] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[14] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[15] S. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelism. *IBM Journal of Research and Development*, 1991.

[16] W. L. Harrison III. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation: An International Journal*, 2(3), 1989.

[17] R. Halstead Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4), 1985.

[18] R. L. Kruse. *Data Structures and Program Design*. Prentice Hall, 1989.

[19] J. R. Larus and P. N. Hilfinger. Restructuring Lisp programs for concurrent execution. In *Proc. ACM SIGPLAN PPEALS – Parallel Programming: Experience with Applications, Languages and Systems*, pages 100–110, July 1988.

[20] S. Moon and M. Hall. Evaluation of predicated array dataflow analysis for automatic parallelization. In *Proc. ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Atlanta, GA, May 1999.

[21] D. Patel and L. Rauchwerger. Principles of speculative runtime parallelization. In *Proc. 11th Workshop on Languages and Compilers for Parallel Computing*, August 1998.

[22] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A framework for exploiting data and functional parallelism on distributed memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 8(11), November 1997.

[23] L. Rauchwerger. Run-time parallelization: It's time has come. *Journal of Parallel Computing*, 24(3-4), 1998.

[24] M. Rinard and P. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, Philadelphia, PA, May 1996.

[25] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *Proc. ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Atlanta, GA, May 1999.

[26] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proc. 1995 International Conference on Supercomputing*, Barcelona, Spain, July 1995.