01 Jan 1989

Expectations for Associative-Commutative Unification Speedups in a Multicomputer Environment

Ralph W. Wilkerson
Missouri University of Science and Technology, ralphw@mst.edu

Bruce M. McMillin
Missouri University of Science and Technology, ff@mst.edu

Recommended Citation

R. W. Wilkerson and B. M. McMillin, "Expectations for Associative-Commutative Unification Speedups in a Multicomputer Environment," *Proceedings of the 13th Annual International Computer Software and Applications Conference, 1989. COMPSAC 89*, Institute of Electrical and Electronics Engineers (IEEE), Jan 1989.
The definitive version is available at https://doi.org/10.1109/CMPSAC.1989.65077

Expectations for Associative-Commutative Unification Speedups in a Multicomputer Environment

Ralph W. Wilkerson and Bruce M. McMillin

Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401

Abstract

An essential element of automated deduction systems are unification algorithms which identify most general substitutions which, when applied to two expressions, make them identical. However, functions which are associative and commutative, such as the usual addition and multiplication functions, often arise in term rewriting systems, program verification, the theory of abstract data types, and logic programming. Unfortunately, the introduction of the associative and commutative equality axioms together with standard unification brings with it problems of termination and unreasonably large search spaces. One way around these problems is to remove the troublesome axioms from the system and to employ a unification algorithm which unifies modulo the axioms of associativity and commutativity. Unlike standard unification, the associative-commutative (AC) unification of two expressions can lead to the formation of many most general unifiers. This paper reports on a hybrid AC unification algorithm which has been implemented to run in parallel on an Intel iPSC/2.

1 Introduction

It has been shown that standard unification is an inherently linear process which does not significantly benefit from parallelization. Over the past 13 years, researchers have been attempting to improve the efficiency of AC unification with special attention to unification problems which arise in term rewriting systems and automated reasoning [*Bu*88]. With the recent development of such systems as parallel Prolog and parallel theorem provers, the necessity of doing AC unification in parallel takes on greater significance. Fortunately, AC unification affords numerous opportunities to exploit parallelism from the basis generation to the calculation of unifiers. Two basic problems must be overcome in finding the complete set of AC unifiers of two expressions. First, a basis of solutions of a homogeneous linear diophantine equation must be determined and second, once the basis of solutions has been discovered, unifiers must be generated through a time consuming search process. The unifiers generated from one solution of the Diophantine equation are independent of any other solution to the equation. Therefore, once the Diophantine equation has been solved, the unifiers can be calculated from the solutions in parallel. In this paper we describe the results of our implementation of AC unification, paying particular attention to unification problems which arise in theorem proving applications. For the purpose of completeness, we have included the necessary background material on AC unification. We begin by stating some basic definitions.

This research has been supported in part by the National Science Foundation under Grant number CDA-8820714, the Intelligent Systems Center at the University of Missouri-Rolla, the McDonnell Douglas Corporation, Intel Scientific Computers, the Missouri Research Assistance Act, and the AMOCO faculty development program.

Variables are designated by the names u, v, w, x, y, z, u_i, v_i, w_i, x_i, y_i and z_i for $i \geq 0$. *Function symbols* are designated by the names $+$, $\times$, f, g, h, f_i, g_i and h_i for $i \geq 0$. *Constants* are designated by the names a, b, c, d, e, a_i, b_i, c_i, d_i and e_i for $i \geq 0$.

A *term* is defined recursively as follows:

(1) A variable is a term.
(2) A constant is a term.
(3) If f is a function symbol and $t_1, \ldots, t_n$ are terms, $f(t_1, \ldots, t_n)$ is a term.
(4) Only those syntactic structures defined by (1)-(3) are terms.

A *substitution* represented by the names θ, λ, σ, θ_i, λ_i and σ_i where $i \geq 0$, is a function mapping variables into terms. It is written $\theta = \{v_1 \leftarrow t_1, \ldots, v_n \leftarrow t_n\}$ where $n \geq 0$. Since a substitution is a mapping, the v_i's are distinct such that $v_i \neq v_j$ for $i \neq j$. The empty substitution is represented by ε. A substitution θ is *applied to* a term t by *simultaneously* replacing every variable in t that is in the domain of θ by the corresponding term. We write $t\theta$ to represent θ applied to t.

2 Unification

Unification is a pattern matching process in which two or more terms are made equal by substitutions of their variables. A set of terms is said to be *unifiable* if there exists a substitution which, when applied to each of them, makes them equal. This substitution is called a *unifier*. A unifier μ is called a *most general unifier* or *mgu* of a set of terms if, for every unifier σ of the set, there exists a substitution λ such that $\mu \circ \lambda = \sigma$. For example, if we let $t_1 = f(x, a)$ and $t_2 = f(y, z)$ then the number of unifiers is infinite and include:

(1) $\{x \leftarrow a,\ y \leftarrow a,\ z \leftarrow a\}$
(2) $\{x \leftarrow b,\ y \leftarrow b,\ z \leftarrow a\}$
(3) $\{x \leftarrow y,\ z \leftarrow a\}$
(4) $\{y \leftarrow x,\ z \leftarrow a\}$

Unifiers (3) and (4) are mgu's and are, in fact, identical

modulo variable renaming. Since both x and y are replaced by the same variable, the name of the variable is arbitrary.

The concept of unification dates back to the introduction of Herbrand's theorem [*He*30]. However, it was not until Robinson [*Ro*65] presented his landmark paper on resolution as a theorem-proving tool that there was an efficient algorithm for finding a unifier. Almost all theorem-proving and term-matching systems to date use Robinson's algorithm or some extension of it.

Let A be a set of terms. We call B the *disagreement set* of A where B is the set of all subterms of the terms in A which begin at the first symbol position at which not all the terms of A have the same symbol. For example, let $A = \{f(x, g(x, y)), f(x, z), f(x, h(a, b))\}$. The disagreement set of A is $\{g(x, y), z, h(a, b)\}$. We can see that the disagreement set of A is empty if and only if A is empty or a singleton. We extend the definition of applying a substitution to a set of terms such that $\{t_1, \ldots, t_n\}\theta = \{t_1\theta, \ldots, t_n\theta\}$. Hence, if θ unifies A, $A\theta$ is a singleton.

Simply stated, Robinson's algorithm begins with an empty unifier μ. If the set of terms A is a singleton, μ is the most general unifier and the algorithm terminates. Otherwise, it scans the terms until it finds a symbol which is not the same in all the terms. It then considers the subterms beginning at this symbol. It chooses two such subterms. (There may or may not be more.) If one of these subterms V is a variable, it checks if the variable occurs in the second subterm U. If it does, there is no unifier and the algorithm terminates. This is called the occurs check. Otherwise, V is replaced by U in A and μ is composed with $\{V \leftarrow U\}$. If neither subterm is a variable, there is no unifier and the algorithm terminates. If the algorithm has not yet terminated, it repeats with A and μ updated. Robinson's algorithm will always terminate and, if a set of terms is unifiable, it will find the most general unifier.

In theorem-proving and term-matching applications, we often work with functions which have properties such as associativity, commutativity, identity or idempotence. For instance, if the function f is commutative, the terms $f(a, x)$ and $f(b, y)$ will not unify under ordinary unification, although the substitution $\{x \leftarrow b,\ y \leftarrow a\}$ will make the terms equal under the commutative property. One solution to this problem is to build into the rule base of the system, rewriting rules that will generate every equivalent expression for the terms with regard to the property or properties belonging to each function. Unfortunately, this strategy may generate an excessive amount of useless clauses which will impede the efficiency of the system.

A more elegant solution is to build these properties into the unification algorithm. This method has the advantage of unifying equivalent terms without having to rewrite the terms in a form in which they unify under ordinary resolution. This greatly decreases the number of intermediate clauses used in solving a problem. A disadvantage is that for every property or set of properties, a new algorithm must be developed. For instance, a separate algorithm is needed for functions which are associative only, commutative only, or both associative and commutative. This type of unification is called *E-unification* where E represents the equations or axioms defining the property or properties. Hence, AC-unification is unification under the associative and commutative properties. In the context of E-unification, ordinary Robinson unification is called null-E unification.

3 AC Unification

Unlike ordinary or null-E unification, E-unification does not guarantee a single mgu. A set of unifiers $\{\theta_1 \ldots \theta_n\}$ is said to be *complete* if for any unifier σ, there exists $1 \le i \le n$ and λ such that $\theta_i \circ \lambda = \sigma$. The set of unifiers is *minimal* if for $1 \le i \le n$, $1 \le j \le n$ and $i \ne j$, there does not exist λ such that $\theta_i \circ \lambda = \theta_j$. It is not uncommon for a minimal, complete set of unifiers of two relatively short terms to contain hundreds or even thousands of unifiers.

Consider the associative and commutative function f. The minimal, complete set of unifiers for the terms $f(x, a)$ and $f(u, b, v)$ is

$$\{v \leftarrow a,\ x \leftarrow f(u, b)\}$$
$$\{u \leftarrow a,\ x \leftarrow f(b, v)\}$$
$$\{u \leftarrow f(a, z_1),\ x \leftarrow f(z_1, b, v)\}$$
$$\{v \leftarrow f(a, z_2),\ x \leftarrow f(u, b, z_2)\}$$

Here, z_1 and z_2 are variables not in the original terms. These are called *introduced variables*.

It should be noted that since f is associative, the term $f(u, b, v)$ in the previous example could have been written as $f(f(u, b), v)$ or $f(u, f(b, v))$. Removing nested function symbols in this manner is called *flattening*.

Stickel [*St*81] presented an algorithm for unifying two terms whose function is associative and commutative. Because associative terms can be flattened, we assume the associative and commutative (AC) function can have an arbitrary number of arguments. AC unification is also known as *bag unification* and can be thought of as unifying two multisets since the terms can be flattened and are order-independent. To unify the terms, they are first flattened and arguments common to both are removed from both terms. Removing the common arguments may eliminate the generation of unifiers that, although correct, are less general than other unifiers. For instance, if the common argument $g(x)$ is not eliminated from the terms, $f(g(x), g(a))$ and $f(g(y), g(x))$ whose most general unifier is $\{y \leftarrow a\}$, unification may result in the additional generation of the unifier $\{x \leftarrow a,\ y \leftarrow a\}$.

Stickel first presents an algorithm for unifying two AC terms whose arguments are all variables. This algorithm, solving the variable-only case, is used by Stickel's general-case algorithm which unifies any two AC terms. In the case that all arguments are variables, to unify the terms $f(x_1, \ldots, x_n)$ and $f(y_1, \ldots, y_m)$ we assign each variable a term of the form t_i whose function symbol is not f or a term of the form $f(t_1, \ldots, t_k)$. For such an assignment to be a unifier, each term t_i must appear an equal number of times in each term. Let

$s_1 = f(x, x, y)$ and $s_2 = f(u, v, v, w)$. $\theta = \{x \leftarrow a,\ y \leftarrow f(b, b),\ u \leftarrow a,\ v \leftarrow b,\ w \leftarrow a\}$ is a unifier of s_1 and s_2 since $s_1\theta = s_2\theta = f(a, a, b, b)$.

Each term t_i in the substitution must conform to the homogeneous linear Diophantine equation

$$\sum_{j=1}^{m} a_j x_j = \sum_{k=1}^{n} b_k y_k$$

in which a_j and b_k represent the number of occurrences of the j^{th} and k^{th} variable in the first term and second terms, respectively, and x_j and y_k represent the number of times t_i appears in the substitution of the j^{th} and k^{th} variable in the first and second terms, respectively.

For instance, the equation corresponding to the terms s_1 and s_2 is $2x_1 + x_2 = y_1 + 2y_2 + y_3$. Nonnegative integral solutions to this equation can be used to represent unifiers since each variable can be assigned a nonnegative integral number of occurrences of each term. Although the number of solutions to a homogeneous linear Diophantine equation is infinite, we can find a finite set of basis solutions such that each solution is a linear combination of these basis solutions. Table I contains the basis solutions to the above Diophantine equation.

Associated with each basis equation is an introduced variable z_i. For each combination of basis equations such that there is at least one nonzero coefficient corresponding to each original variable, we

Table I. BASIS OF SOLUTIONS TO DIOPHANTINE EQUATION

x	y	u	v	w	
0	1	0	0	1	z_1
0	1	1	0	0	z_2
0	2	0	1	0	z_3
1	0	0	0	2	z_4
1	0	0	1	0	z_5
1	0	1	0	1	z_6
1	0	2	0	0	z_7

can construct a unifier. The term replacing each variable is made of the introduced variables associated with the basis equations. The coefficient corresponding to an original variable and an introduced variable determines the number of times the introduced variable is represented in the term replacing the original. For instance, the unifier generated from basis equations 3, 4 and 6 is $\{x \leftarrow f(z_4, z_6),\ y \leftarrow f(z_3, z_3),\ u \leftarrow z_6,\ v \leftarrow z_3,\ w \leftarrow f(z_4, z_4, z_6)\}$. The variable-only algorithm is shown more formally in figure 1.

To find the unifiers for AC terms with arbitrary arguments (which may be AC functions, ordinary functions, constants or variables) we create two new terms called the *variable abstraction* of the original terms by replacing each distinct argument with a new variable. For instance, the variable abstraction of $f(a, a, x)$ and $f(y, y, b)$ is $f(x_1, x_1, x_2)$ and $f(y_1, y_1, y_2)$ with the substitution $\{x_1 \leftarrow a,\ x_2 \leftarrow x,\ y_1 \leftarrow y,\ y_2 \leftarrow b\}$.

1. Eliminate common terms.

2. Form an equation from the two terms where the coefficient of each variable in the equation is equal to the multiplicity of the corresponding variable in the term.

3. Generate a basis of nonnegative integral solutions to the equation.

4. Associate with each solution a new variable.

5. For each sum of the solutions (no solution occurring in the sum more than once) with no zero components, assemble a unifier composed of assignments to the original variables with as many of each new variable as specified by the solution element in the sum associated with the new variable and the original variable.

Figure 1. Stickel's Variable-Only AC Unification Algorithm

We next use the variable-only algorithm to find the unifiers to the variable abstraction. For efficiency, we introduce additional constraints for generating the variable-only unifiers. Any unifier which assigns a nonvariable to an argument corresponding to a nonvariable in the original terms is eliminated. Likewise any unifier which assigns the same variable to two arguments corresponding to arguments in the original terms that obviously will not unify are discarded. In the above example the unifiers are

(1) $\{x_1 \leftarrow z_4,\ x_2 \leftarrow z_1,\ y_1 \leftarrow z_4,\ y_2 \leftarrow z_1\}$
(2) $\{x_1 \leftarrow z_4,\ x_2 \leftarrow f(z_1, z_2, z_2),\ y_1 \leftarrow f(z_2, z_4),\ y_2 \leftarrow z_1\}$

The last step is to unify each of these unifiers with the substitution corresponding to the variable abstraction. In this example, this is the unifier $\{x_1 \leftarrow a,\ x_2 \leftarrow x,\ y_1 \leftarrow y,\ y_2 \leftarrow b\}$.

This gives us the following results.
(1) $\{x \leftarrow b,\ y \leftarrow a\}$
(2) $\{x \leftarrow f(b, z_2, z_2),\ y \leftarrow f(z_2, a)\}$

Figure 2 contains Stickel's general AC unification algorithm. Note that this algorithm may be called recursively in step 3 for terms with AC functions and that Robinson's unification algorithm is called for all other terms.

Stickel's AC unification algorithm requires a basis of solutions for a homogeneous linear Diophantine equation with integer coefficients of the form

$$\sum_{i=1}^{m} a_i x_i = \sum_{j=1}^{n} b_j y_j .$$

Several algorithms have been developed to generate such a basis. Huet [*Hu*78] developed an algorithm which begins with the trivial (all zero) solution and generates basis solutions by enumeration. His

algorithm determines bounds which provide stopping conditions for the enumeration.

Huet proves that any solution, such that some x_i is greater than the largest b_j or likewise some y_j is greater than some a_i, is nonminimal. Huet also shows that for any $1 \le i \le m$ and $1 \le j \le n$, where $\mathbf{lcm}_{ij}$ is the least common multiple of a_i and b_j, a solution such that $x_i = \mathbf{lcm}_{ij}/a_i$ and $y_j = \mathbf{lcm}_{ij}/b_j$ and all other coordinates

1. Form generalizations (the variable abstraction) of the two terms by replacing each distinct argument by a new variable.

2. Use the algorithm for the variable-only case to generate unifiers for the generalizations of the two terms. The variable-only-case algorithm may be constrained to eliminate the generation of unifiers assigning more than one term to variables whose value must be a single term, and the generation of unifiers which will require the later unification of terms which are obviously not unifiable.

3. Unify for each variable in the substitution from step 1 and the unifiers from step 2 the variable values and return the resulting assignments for variables of the original terms. This is the complete set of unifiers of the original terms.

Figure 2. Stickel's General AC Unification Algorithm

0, is minimal. Therefore, any solution in which all coordinates are greater than or equal to the respective coordinates of any of these solutions is nonminimal. Using these constraints, Huet constructs bounds to limit the enumeration. Each potential solution within these bounds is checked whether it is a solution and that it is not greater than any solution already found.

Lankford [*La*87] developed an algorithm which uses elementary row operations on a matrix to generate a basis of solutions. By keeping the matrix irredundant, Lankford ensures the basis formed is minimal.

Lankford represents the homogeneous equation as

$$\sum_{i=1}^{m} a_i x_i - \sum_{j=1}^{n} b_j y_j = 0.$$

The *norm* of an $m+n$-tuple S is defined as

$$\|S\| = \sum_{i=1}^{m} a_i s_i - \sum_{j=1}^{n} b_j s_{m+j}.$$

We define A to be the set of all $m+n$-tuples S such that $1 \le i \le m$, $s_i = 1$ and all other coordinates are 0. Likewise, B is the set of all $m+n$-tuples S such that $m+1 \le i \le m+n$, $s_i = 1$ and all other coordinates are 0.

Lankford's algorithm iteratively finds the sets X^k, P^k, N^k, and Z^k. The initial conditions are

X^1 = the empty set,
$P^1 = A$,
$N^1 = B$,
Z^1 = the empty set.

The inductive definition of the subsequent generations is

$$X^{k+1} = (A + N^k) \cup (B + P^k),$$

$$P^{k+1} = \{S \mid S \in X^{k+1}, \|S\| > 0, \text{ and } S \text{ is irreducible relative to } Z^k\},$$

$$N^{k+1} = \{S \mid S \in X^{k+1}, \|S\| < 0, \text{ and } S \text{ is irreducible relative to } Z^k\},$$

$$Z^{k+1} = Z^k \cup \{S \mid S \in X^{k+1} \text{ and } \|S\| = 0\}.$$

In the above definition, S is reducible relative to Z^k if there exists some $Z \in Z^k$ such that each coordinate of X is greater than or equal to the corresponding coordinate of S.

The algorithm terminates when P^k and N^k are empty. When this occurs, Z^k contains an irredundant basis.

Zhang [*Zh*87] developed a very efficient algorithm which finds the basis solutions to a homogeneous linear Diophantine equations in which several coefficients are 1's. In practice, many of the Diophantine equations appearing in AC unification problems are of this form. The simple case solves Diophantine equations in which all the coefficients on one side of the equation are 1's. This algorithm has the additional asset that intermediate results can be stored and need not be recalculated every time they are needed. The general case solves equations which have two or more 1-coefficients regardless of where they are in the equation. Zhang's algorithm reduces the equation to smaller equations with only one 1-coefficient and uses some other algorithm (possibly Huet's or Stickel's) to solve them. The smaller equations will generally require less time to solve than the original equation.

Zhang's simple case algorithm considers Diophantine equations of the form

$$\sum_{i=1}^{m} x_i = \sum_{j=1}^{n} b_j y_j.$$

Zhang defines a set $C(m,k) = \{(k_1, \ldots, k_m) \mid k_1 + \cdots + k_m = k\}$ and a vector e_n^j which is a vector of length n such that all components are 0 except the j^{th} component is 1. The basis of the Diophantine equation is the set of vectors $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$ such that $1 \le j \le n$, $X \in C(m,j)$, and $Y = e_n^j$. Once $C(m,j)$ is computed, it can be stored and used in solving other equations. Zhang uses the same concept to simplify finding the basis solutions to Diophantine equations having more than one 1-coefficient, with neither side containing all 1-coefficients. For the details of this algorithm see [*Zh*87].

Zhang's simple-case algorithm is used to construct the basis solutions to the Diophantine equations when the coefficients on one side of the equation are all 1's. Otherwise, Lankford's algorithm is used.

4 Implementation Details

Terms are represented as trees where each node is a function symbol, constant, or variable. Only function symbols have children. The nodes are represented by type (function, constant, or variable) and an integer identifier. The identifiers are associated with character string names by a table. The names are used only for the user interface. Each node has a pointer to its first child, if any, and its next sibling. A flattened term is represented as a linked list of pointers to the term's arguments. The AC function symbol is not a part of the data structure. This data structure allows the term to appear flat without changing any of the pointers in the term or having to make a copy of the subterms. Unifiers are represented as linked lists, where each list element consists of a variable identifier and a pointer to a term. A set of unifiers is a linked list of pointers to unifiers.

Substitutions are not made directly to the terms. Instead, whenever a node is inspected, and the node represents a variable, the node's identifier is compared to the variable identifiers in the substitution. If that identifier is found, the term pointed to by the substitution is used in place of the variable. If this new term is itself a variable, this process is repeated until either a nonvariable or a variable not in the substitution is reached. If the substitutions were applied to the terms, the terms could grow exponentially in size as multiple occurrences of variables are replaced with terms which may themselves contain repeated variables requiring substitution.

Robinson's algorithm had to be modified slightly to work in conjunction with AC functions. It is implemented recursively such that if the two terms to be unified are the same function and have the same number of arguments, the arguments are unified by a generic unification routine. This generic routine determines if the terms are the same AC function and, if they are, it uses Stickel's algorithm, otherwise it uses Robinson's.

Robinson's algorithm was also modified to handle more than one unifier. Since the arguments of the terms may contain AC functions, more than one substitution may be returned by the generic unification routine. Subsequent arguments are unified independently for each of these substitutions. Substitutions resulting from these unifications are accumulated and likewise used in unifying subsequent arguments. Hence, this algorithm may return multiple unifiers.

To prevent the occurs check from redundantly searching the same subterms for an occurrence of the same variable, a list is maintained of variables whose substitutions do not contain the target variable. The first time a variable is encountered that is in the substitution list, the associated term is searched. If the target variable is not found, the encountered variable is put on the list of searched variables. Otherwise, the occurs check fails. If a variable is encountered that is on the list of searched variables, it is immediately skipped.

Our implementation of of Stickel's algorithm varies from that presented in earlier. Rather than determining the complete set of unifiers for the variable abstraction and then unifying these with the substitution defining the variable abstraction, we create each variable-only unifier as we need it and unify it with the variable-abstraction substitution. It should be apparent that the variables representing the variable abstraction need not be used in either scheme. It is only the terms associated with these variables that are actually unified.

We implemented the parallel version of the AC unification algorithm to run on an Intel iPSC/2. The master process runs on one node of the cube and a slave runs on the other nodes of the cube. The processes communicate via message passing.

Our unification program presents several possible opportunities to exploit parallel processing. For instance, the occurs check in Robinson's unification algorithm need not be sequential. Recall that the occurs check checks for an occurrence of some variable in a term. If the term being checked is a function with several arguments, each argument can be checked for the variable by a different processor. However, the overhead of message passing far outweighs the processing needed to perform the occurs check on one term. Hence, the grain size of the occurs check is too small to make efficient use of parallelism.

The algorithms used to find the basis solutions to the Diophantine equations similarly can be designed to distribute the processing, but again, the grain size of the problem is rather small compared to the overhead inherent in our message-passing scheme.

Since our implementation of Robinson's algorithm allows multiple substitutions to be returned when unifying a terms arguments, subsequent arguments must be unified using each of the previous substitutions. Each of these unifications may be done by a different processor. Consider the terms $f(+(x,y),+(x,z))$ and $f(+(a,u),+(b,v))$, where $+$ is AC and f is not. Robinson's unification algorithm would first unify $+(x,y)$ and $+(a,u)$ using the AC unification algorithm. Stickel's algorithm will return four substitutions. The next pair of arguments are unified four times, once for each of these substitutions. These unifications may be each be distributed to a separate processor.

Parallelism can be exploited similarly in the last step of Stickel's algorithm when the subterms corresponding to the variable abstraction are unified with the introduced-variable terms. Another way to exploit parallelism in Stickel's algorithm is to generate the unifiers corresponding to each particular solution of the Diophantine equation on a separate processor. This method is appealing since the generation of the unifier involves unifying all the arguments of the terms. The grain size of the distributed subtasks is larger than

in the previous methods in which each distributed subtask unified one argument.

We decided to implement our program using this last plan. Our AC unification algorithm runs sequentially up to the point where the valid solutions to the Diophantine equation are determined. The master process then sends a solution to each slave, which finds the unifiers associated with that solution.

The master process contains all of the elements of our sequential implementation with the exception of the sequential AC unification algorithm itself. The user interface and Robinson unification algorithm remain unchanged. The mainline was modified to create and terminate the slave processes.

The slave processes consist of the distributed part of the AC unification algorithm and all routines necessary to unify the subterms. If additional unification is required in the slaves, either ordinary unification or AC unification, it is done sequentially and not in parallel.

The processes exchange the following types of messages.

```
AC_unify {
   Flatten both terms.
   Remove arguments common to both terms.
   For each distinct argument {
      Add it to variable abstraction.
      Determine its multiplicity.
   }
   Form Diophantine equation from multiplicities
    of arguments.
   Find basis of solutions to Diophantine equation.
   Remove illegal basis vectors.
   Find all valid solutions to Diophantine equation.
   If no solutions exist
      Return(fail).
   Send TERMS to slaves.
   UNIFIERS = empty list.
   NUM_DONE = 0.
   Send several problems to each slave.
   While (NUM_DONE < number of slaves) {
      Receive message from slave.
      If (message = = SOLUTION) {
         Add solution to UNIFIERS.
         If (more problems to send)
            Send more problems to slave.
         Else
            Send DONE to slave.
      }
      Else if (message = = DONE)
         NUM_DONE = NUM_DONE + 1.
   }
   If (UNIFIERS = = empty list)
      Return(fail).
   ELSE
      Return(success).
}
```

Figure 3. AC Unification Algorithm - Master

TERMS - sent from the master to the slaves. Contains the terms and a substitution representing the variable abstraction of the arguments of the terms being unified and the substitution calculated so far.

PROBLEM - sent from the master to a slave. Contains the basis vectors that make up a particular solution to the Diophantine equation.

SOLUTION - sent from a slave to the master. Contains unifiers corresponding to the problem sent to the slave.

Figures 3 and 4 contain pseudocode for the master and slave components of the AC unification algorithm.

```
slave {
   message = READY.
   While (message != END_SIGNAL) {
    Receive message from master.
    If (message = TERMS) {
       Parse terms from message.
       Parse SUBSTITUTION from message.
       Send READY to master.
       message = PROBLEM
       While (message = = PROBLEM) {
        Receive message from master.
        If (message = = PROBLEM) {
           Determine variable-only unifier.
           OLD_UNIF = SUBSTITUTION.
           For each term in variable abstraction {
            NEW_UNIF = empty.
            For each substitution in OLD_UNIF {
             Unify variable-only,
                  variable-abstraction terms.
             Append unifier (if any) to NEW_UNIF.
            }
            OLD_UNIF = NEW_UNIF
           }
           While (Too many partial unifiers for
                one message) {
            Send SOLUTION to master.
           }
           Send Solution to master.
         }
         Else if (message = = DONE)
           Send DONE to master.
       }
    }
   }
   Send END_SIGNAL
}
```

Figure 4. AC Unification Algorithm - Slave

Table II shows the running time comparison between the two implementations in seconds of wall time for some representative problems. The functions + and * are both associative and commutative.

Table II. AC UNIFICATION TIMES USING FIVE SLAVES

Problem	Terms being unified	# sols	seq time	par time
(1)	+(x,x,y) +(u,v,v,c)	18	.06	.203
(2)	+(x,y,z) +(u,v,w,xx)	2161	9.34	3.71
(3)	+(*(a,a,x,x),*(b,c,y,y,z),*(a,b,c,x)) +(*(a,b,u),*(c,c,u,u),*(c,u,v))	51	1.05	.58
(4)	+(x,*(x,y),*(y,z)) +(*(u,v),*(v,v,a),u)	1610	11.74	14.39
(5)	+(*(a,x,y),*(b,xx,yy),*(c,yx,yy)) +(*(d,u,v),*(e,uu,vv),*(d,e,f))	0	315.3	76.4
(6)	+(x,x,x) +(u,v,w,c)	6006	24.2	10.8

Table III. AC UNIFICATION TIME BREAKDOWN

Problem	S_N	S_L	S_T	R_N	R_L	R_T	*Prep*	*Compile*	*Slave*	*Unify*
(1)	16	1336	.004	16	5948	.011	.041	.009	.015	.003
(2)	25	71564	.037	128	5.47E6	.660	.956	1.65	1.90	.832
(3)	16	2780	.006	21	.120E6	.320	.012	.010	.415	.379
(4)	16	2264	.007	232	4.26E6	12.28	.058	2.97	10.5	8.66
(5)	16	2580	.006	16	0	76.15	.069	0	76.19	73.62
(6)	45	.172E6	.087	403	15.E6	.540	3.04	6.33	4.56	1.80

5 Expectations for the Multicomputer Environment

The results indicated by Table II show mediocore parallel performance. Additional instrumentation of the processes showed where the time was being spent. This data is tabulated in Table III for five slave processes where S_N is the number of sends, S_L is the length of all sends S_T is the total time for all sends with receives R_N, R_L, R_T similarly defined. Prep is the time to find the solutions to the Diophantine equation and to send the terms to the slaves. Compile is the overhead associated with receiving the results and managing the received buffers. Slave is the maximum total time spent by the five slaves and Unify is this time exclusive of the buffer management overhead.

As is to be expected, very small problems, such as problem (1), perform worse in the parallel environment than in the sequential environment, due to the overhead of message passing in distributing/collecting the results. Larger problems show a modest speedup as shown in Figure 5, limited primarily by the sequential collection of results in "compile". We expect this can be performed in parallel in $\log_2 n$ steps using n processors via a tree-reduction method. Problem number 4 also shows shortcomings in the implementation. Problem number 4 requires a

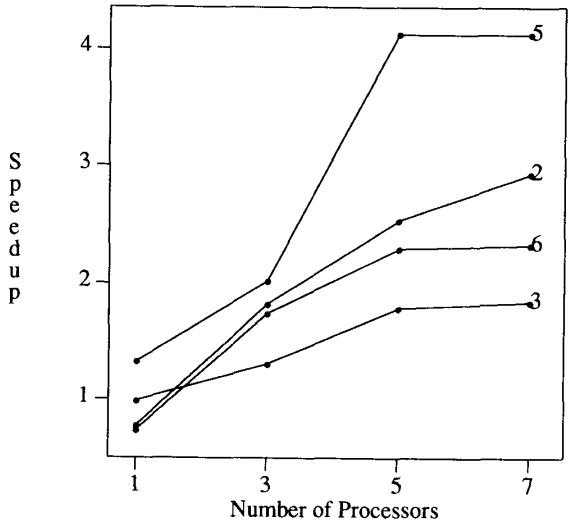

Figure 5. Parallel Speedup.

substantial amount of additional unification of subterms in the slave which, due to our implementation, is done sequentially. This, plus the overhead associated with message passing makes the parallel version run slower than the sequential version. The best speedup, obtained from problem 5, results from the decomposition of the problem into six parts which also accounts for the limiting value of the speedup between 5 and 7 slaves. This work demonstrates that AC unification is amenable to parallel speedup. For problems with a large number of subproblems to be solved, such as in problems 2 and 6, the potential for massive parallel speedup exists.

The significance of this result is that in an actual theorem proving application, we will be performing thousands of these problems. It is possible that these problems may be batched such that slave-host communication is reduced. Furthermore, some subproblems require considerable more work than others, consequently if we could determine those problems before distributing the workload, efficiency could be increased. The major problems encountered in our tests were the common difficulties of estimating the complexity of each slave task and the bottleneck formed by sequentially compiling the results. We are looking at ways of estimating the former and of implementing the latter efficiently. The end goal of this research is to embed this parallel unification function in a parallel theorem-proving environment.

References

[*Bu88*] Burckert H., Herold A., Kapur D., Siekmann J., Tepp M., "Opening the AC-unification race.", *Journal of Automated Reasoning,* vol 4 (1988), pp. 465-474.

[*CL87*] Christian J., and Lincoln P., "Adventures in associative-commutative unification." MCC Technical Report ACA-ST-275-87, 1987.

[*Fo87*] Fortenbacher A., "An algebraic approach to unification under associativity and commutativity." *Journal of Symbolic Computation,* vol 3, (1987) pp. 217-229.

[*He30*] Herbrand J., "Recherdies sur la Theorie de la Demonstration." *Trauaux de la Societe des Sciences et Lettres de Varsovie, Classe III Sci. Math. Phys.,* 33, 1930.

[*Hu78*] Huet G., "An algorithm to generate the basis of solutions to homogeneous linear Diophantine equations." *Information Processing Letters,* vol 7, no 3, (April, 1978) pp. 144-147.

[*La87*] Lankford D.S., "Non-negative integer basis algorithms for linear equations with integer coefficients." (unpublished), 1987.

[*Re87*] Reed, D. and Fujimoto, R. *Multicomputer Networks: Message-Based Parallel Processing,* The MIT Press, Cambridge, MA, 1987.

[*Ro65*] Robinson J.A., "A machine-oriented logic based on the resolution principle." *Journal of the Association for Computing Machinery,* vol 12, no 1, (January, 1965), pp. 23-41.

[*St81*] Stickel M., "A unification algorithm for associative-commutative functions." *Journal of the Association for Computing Machinery,* vol 28, no 3, (July, 1981), pp. 423-434.

[*Zh87*] Zhang H., "An efficient algorithm for simple Diophantine equations." Technical Report 87-26, Department of Computer Science, Rensselaer Polytechnic Institute, Troy, New York. 1987.