Constant Plurality: A New Concept for Automated Parallelisation

J. M. MacLaren
Centre for Novel Computing,
Department of Computer Science,
University of Manchester,
Oxford Road,
Manchester M13 9PL,
UK
email: jon.maclaren@cs.man.ac.uk

Abstract

Parallelising compilers try to uncover implicit parallelism in sequential programs, generating a semantically equivalent parallel program. Typically, these compilers focus on detecting implicit parallelism in loop nests, using techniques such as loop transformation *to expose parallelism. When compiling for single address space machines,* privatisation, *which creates multiple copies of a data structure in a parallel program, is one of the most useful of these transformations. This paper reconsiders privatisation, and hypothesises that data structures should be privatised because of what they represent, rather than because of how they are accessed in loop nests. A data structure's* plurality *is defined as indicating whether or not that data structure has been privatised. A compilation method is described which fixes the plurality of each data structure in the target program before parallelisation begins. An implementation of this method, MRes, is used to parallelise a number of benchmarks, including the SPEC CFP95 suite; an evaluation shows that six of the fifteen codes can be effectively parallelised with every variable having a fixed plurality.*

1. Introduction

State-of-the-art automated parallelisation is still limited [11]. Many leading research parallelisers only search for implicit loop parallelism. These parallelisers analyse the loop nests in the program; *loop transformations* may be employed to expose implicit parallelism. Research by the Polaris group has found the most useful loop transformation for enabling parallelisation to be *array privatisation* [3], when compiling for Single Address Space machines.[1]

```
DO I=1,N
   X    =A(I)+B(I)
   C(I)=X**2+5*X+4
ENDDO
```

Figure 1. Loop using a temporary scalar.

When a variable is *privatised* in a parallel program, each thread is provided with its own copy of the variable; variables that are not privatised remain *shared* between threads.[2] Privatising temporary variables in a loop nest can break dependences, permitting parallelisation. In Figure 1, privatising `X` will allow the loop iterations to be executed concurrently.

Many scientific applications are engaged in some form of simulation; they model some part of the world that is hard to observe, or to predict accurately, in real life—Numerical Weather Prediction and the modelling of molecular interactions based on quantum phenomena are examples. In such programs, large data structures are used to represent the simulated domain and the objects located therein. The program transforms these data structures, typically simulating a change over time, based on formulae that model interactions in this arena—be they convection and condensation, or quantum tunnelling. The application of these formulae over the entire domain is clearly a repetitive, and thus intuitively iterative, task, which will be im-

[1] A Single Address Space machine is one where threads of execution within a process see the same virtual address space. Such machines may have physically distributed memories.

[2] Privatisation is related to expansion, which creates one copy of the variable for each loop iteration [7].

plemented using loops. To parallelise such codes on a Single Address Space architecture, the data structures representing the simulated domain are *shared* between the multiple threads of execution, which cooperatively transform them. During these transformations, other data structures are used, e.g. to hold the results of temporary calculations; each thread should have its own *private* copy of such structures.

Assuming that each data structure has a clear purpose, then each data structure should remain either shared or privatised for the entire duration of its existence. The term *plurality* is defined for a data structure to be "shared" if there is a single copy of the data structure shared between the threads of execution, or to be "privatised" if each thread has its own private copy; no other values are permitted. It is important to note that plurality decisions are intended to be made by an expert in the target code, rather than an expert in parallelism.

> **Hypothesis (Constant Plurality)** *There exists more than one scientific program that can be effectively parallelised when each of the code's data structures has a constant plurality.*

It is important to note that the hypothesis assumes that, for all programs, each data structure is a separate entity, used for a single purpose, and is capable of assuming its own plurality. The hypothesis cannot be applied to codes that violate this assumption.

To allow the hypothesis to be verified experimentally, a novel source-to-source compilation method has been developed; at the beginning of compilation, the plurality of every variable is fixed. An overview of this method is given in Section 2. The method was implemented as a prototypic compiler, designed to interact with a parallelisation expert when it cannot automatically resolve information about the target program; the implementation is described briefly in Section 3. This prototype was used to assist the parallelisation of two suites of benchmark codes, as described in Section 4. The performance of each parallel code is measured, and used to help judge which of the benchmarks have been effectively parallelised. Codes not effectively parallelised by the prototype are considered further. Related work is considered in Section 5; conclusions are presented in Section 6.

2. Compilation Method

The Constant Plurality Hypothesis can be applied to both of the common code generation styles for Single Address Space machines, i.e. the fork/join and the SPMD styles [5]. In the fork/join model, threads are created (or forked) when required, i.e. to cooperatively execute parts of a parallel loop; in the SPMD model, a number of threads are started at the beginning of the program and remain until the end. In the fork/join model, privatised variables exist only for the duration of a parallel loop; they are copied during loop startup, and killed when the additional threads exit. Constant Plurality is more useful to SPMD, where privatised variables exist at exactly the same time on all threads of execution. For this reason, the parallelisation process described here generates SPMD code.

The method described here is designed to parallelise sequential Fortran 77 programs,[3] and performs source-to-source compilation, i.e. the generated parallel program is also written in Fortran 77. In order to allow multiple threads of execution to cooperate, any implementation must provide the following:

- bootstrapping code which forks multiple threads of execution (with a shared address space) before calling the main program; and
- a mechanism by which these threads can perform barrier synchronisations.

There must also be the notion of a *master thread*, which performs any tasks that must be carried out by a single thread only. Each thread of execution must be able to:

- uniquely identify itself,
- discover the number of threads that are executing the program, and
- detect whether or not it is the master thread.

An overview of the compilation process is given in Figure 2. Following an initialisation phase, the plurality of each of the program's arrays is determined; scalar variables will all be privatised. The fact that Fortran 77 does not support dynamic data allocation makes this task simpler, as all the data structures of a program can be identified from a simple, static analysis. As plurality is based on algorithmic knowledge of the program, it cannot be obtained automatically; a user must contribute this information.

The compilation method exploits only loop parallelism, so the SPMD parallel code will have the same control flow structure as the sequential code, except inside parallelised loops. As shown in Figure 3, the sequential program is broken down into a hierarchy of

[3] Although it is an old language, many scientific application codes in use today are written in Fortran 77, and it is still the language most widely supported by automatic parallelisers.

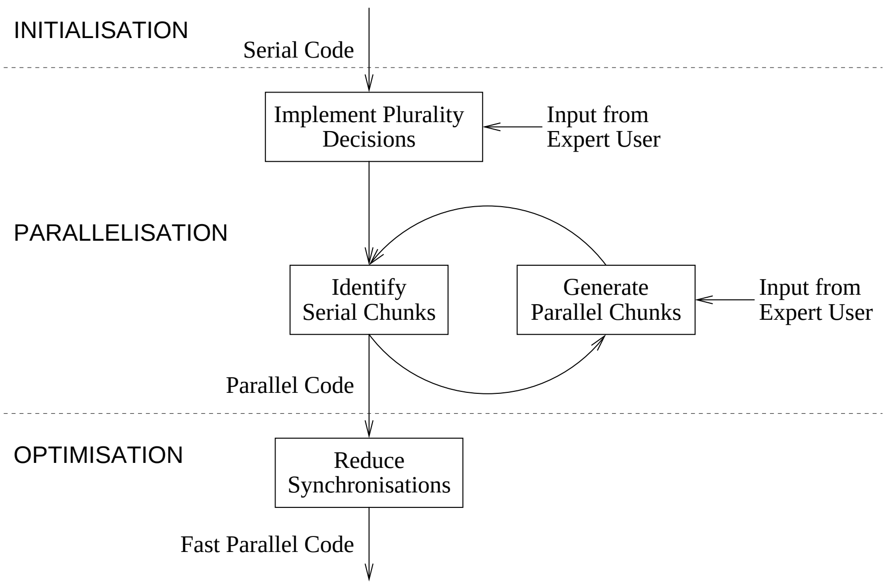

Figure 2. Schematic for compilation process.

chunks;[4] these are converted to parallel code individually. As control flow is preserved, the chunks in the parallel code are executed in the same order as those in the sequential code. Therefore, as long as the parallel chunks modify the program's data structures in the same way as the sequential chunks, the sequential and parallel programs will be equivalent, i.e. they will generate the same results for all possible inputs.[5] Following a detailed description of this parallelisation process, in the following section, techniques for optimising the generated SPMD code are explored.

Note that, in Fortran 77, certain uses of `EQUIVALENCE` and/or `COMMON` declarations can cause different variables to overlap in memory, thus preventing each data structure from being assigned its own plurality. The presented compilation method assumes that no such "illegal" declarations are present.

2.1. Parallelisation

The first step in parallelisation is to alter the program's data structures so as to implement the plurality decisions mentioned above, producing a modified sequential program that is ready for parallelisation. Care must be taken to ensure that, where privatised variables are initialised *before* the program begins, i.e. with `DATA` statements, each thread's copy is initialised in the same way.

The next step is to divide the executable statements of the sequential program into chunks.[6] Chunks are divided into three distinct categories, based on the type of work they perform:

cooperative chunks, one per parallelised loop nest, where the threads in the parallel chunk execute complementary parts of the sequential chunk;

all-thread chunks, which modify privatised variables or alter program control flow, and will be executed by every thread;[7] and, finally,

one-thread chunks, that modify shared variables, or perform I/O,[8] and are executed only on the master thread.

To permit all statements in the program to be unambiguously classified, some statements require rewriting. More precisely:

[4] This term is so named to reflect the fact that code "chunks" may vary considerably in size, sometimes being larger than a "block", sometimes as small as a single statement.

[5] Note that the correctness of the input sequential program for its purpose is irrelevant to the process of parallelisation.

[6] The statements of a Fortran 77 program can be divided into executable statements and nonexecutable (or declarative) statements; lists of these are given in the standard [1, Sec. 7].

[7] The only exceptions are calls to external procedures, which may be classified as one-thread or all-thread.

[8] As I/O is inherently sequential, all I/O in the parallel version of the program should occur in the same order as in the sequential version. The simplest way to achieve this is to execute all I/O on the same thread. This approach is also the most portable as, on some platforms, it may not be possible to share file handles between threads.

```
      PROGRAM DEMO
      INTEGER I
      REAL A(10),MA
      DATA MA/0.0/A/...
C
C     START OF EXEC STATEMENTS
C
      PRINT*,'BEGINS'
      DO 10 I=1,9
         PRINT*,MA
         A(I)=A(I)+A(I+1)+...
         IF(A(I).GT.MA)THEN
            MA=A(I)
         ENDIF
   10 CONTINUE
C
      STOP
      END
```

Figure 3. The hierarchical division of a simple program into chunks.

- Function invocations are removed to separate statements;
- Logical `IF` statements are rewritten as block `IF`s;
- All `ELSE` and `ELSEIF` clauses in block `IF`s are rewritten as separate `IF` statements;
- Each `DO` loop is given its own terminating `CONTINUE` statement; and
- Any labelled executable statements which are not `CONTINUE` statements are split into a labelled `CONTINUE`, and the original executable statement.

Following rewriting, the key source of ambiguity that remains is that `DO` loops may be classified as either cooperative or all-thread. The classification of a `DO` loop is based solely upon whether or not it is to be parallelised: if it is, the loop will form a cooperative chunk, which will include *only* the loop and the statements within it; non-parallelised loops are treated as single all-thread statements, in the way described earlier. As well as analysing the `DO` loop for parallelism, an implementation is also free to transform the loop being classified, even if that loop is then classified as all-thread.[9] As variable pluralities are fixed before parallelisation, further scalar and array privatisation may not be considered.

The bodies of statements containing internal structure, i.e. `IF` statements and all-thread `DO` loops, are divided into a separate set of chunks; the chunks are arranged hierarchically.[10] Chunks that contain sub-chunks are known as *container chunks*; as control flow from chunk to chunk through the parallel program must be the same on all threads, only all-thread chunks may be container chunks. This scheme allows statements containing internal structure to be considered as a single all-thread statement when dividing the program into chunks. Program units are also considered to have a single all-thread container chunk representing the whole unit, and so calls to user program units,[11] are classified as all-thread statements. This scheme can be extended to support explicit branches, e.g. `GOTO` statements, which are non-hierarchical; this is explored in [10, Ch. 3]. Such statements are classified as all-thread statements.

To ensure that all threads of execution view shared variables consistently, synchronisations will thus be placed at every chunk boundary; techniques for relaxing this constraint are discussed later, in Section 2.2.

Parallelisation points are used to help implement the parallelisation process; each parallelisation point is a marker in the program being parallelised. Initially, a single parallelisation point is inserted at the start of the main program, immediately before the first executable statement. The sequential code following the

[9]This allows the use of loop transformations to expose parallelism in an inner-loop.

[10]This scheme assumes that the source program is not recursive. This is not a restriction for Fortran 77, but the method would require modification before application to recursive programs.

[11]A user program unit is any program unit (i.e. main program, subroutine or function) for which the source code is available.

parallelisation point is examined, and a chunk is identified; a parallel chunk is produced to replace it, and the parallelisation point is moved to the end of the new parallel chunk. The sequential program is thus traversed, and transformed into a parallel program, one chunk at a time. If a chunk contains statements with their own substructure, i.e. DO or IF statements in Fortran 77, additional parallelisation points are inserted to help parallelise the "insides" of these statements. Calls to user program units are treated in the same way; an initial parallelisation point is inserted into the called program unit, which is then added to the list of program units requiring parallelisation. Parallelisation points are removed when they reach the end of their container chunk (e.g. the initial parallelisation point stops just before the END statement at the end of the main program). As stated above, container chunks are always all-thread. The division of a simple program into a hierarchy of chunks is shown in Figure 3. Where multiple parallelisation points exist in a program unit, the selection of which parallelisation point to advance is arbitrary. This parallelisation process is described in pseudocode in Figure 4.

2.1.1. Cloning User Procedures

As stated above, calls to user program units are treated in the same way as DO loops and IF statements, i.e. the call is treated as a single all-thread statement.[12] So, if the called program unit has not already been parallelised, a new parallelisation point will be added to it, and the program unit parallelised at a later time. This assumes that only one version of the called program unit will be required. Consider the program in Figure 5; the array LRG is shared, and the array SML is privatised. Both matrices are initialised by calling the same subroutine, FILLER; the initialisation of LRG should be done in parallel, but each thread must initialise their own copy of SML. When parallelising FILLER, the plurality of the argument ARR will be unknown. This could be solved by allowing for the variance in plurality in the parallel code, by using a conditional statement to select between code for shared ARR and privatised ARR at run-time. However, this approach would require a rethinking of the statement classification scheme described earlier, which assumes that the plurality of every variable is known.

A better approach is to create a version, or clone, of FILLER for each call scenario as it is encountered; in general, a separate clone should be used for each combination of shared and privatised arguments.[13] The parallel clones will be kept separate from the original sequential version, which will be required if the program unit is called from within a loop which is parallelised. When parallelising the clones, the plurality of all variables will be known, as is the case for the main program.

```
      PROGRAM CONFL1
      REAL LRG(256,256)
      REAL SML(4,4)
      ...
      CALL FILLER(LRG,256,0.0)
      CALL FILLER(SML,4,1.0)
      ...
      STOP
      END

      SUBROUTINE FILLER
     +     (ARR,SIZE,VAL)
      INTEGER SIZE,I,J
      REAL ARR(SIZE,SIZE),VAL
      DO 10 J=1,SIZE
         DO 20 I=1,SIZE
            ARR(I,J)=VAL
   20    CONTINUE
   10 CONTINUE
      RETURN
      END
```

Figure 5. Subroutine showing argument with differing plurality.

This cloning scheme also implies that the calling procedure knows the identity of the procedure being called; where the called procedure was passed in as an argument, multiple identities are possible.[14] To circumvent this problem, another level of cloning must be added, creating different versions of procedures for each possible value of such arguments. As parallelisation proceeds from the main program unit inwards, the actual name of the procedure being passed in would always be known, even where program units are passed down further into the call tree. While this scheme could, in principle, result in unacceptable growth in size of the code, in all practical examples encountered by the author, the number of procedures passed in this way has

[12] It is now assumed that each program unit has a single entry point and single, standard return point.

[13] Theoretically, for an n-argument program unit, this scheme could create 2^n clones. In practice, most arguments will have the same plurality for every call site, as with the second argument to FILLER in the given example.

[14] The passing of subroutines and functions as arguments is often used by solvers of systems of equations, e.g. ordinary differential equations, where the solver invokes the supplied code as required.

```
INSERT a first parallelisation point into the start of the main program unit
ADD the main program to the list of program units to parallelise
while there are program units to parallelise do
  SELECT a program unit, PU
  while parallelisation points exist in PU do
    SELECT a parallelisation point, PP
    if PP has reached the end of its container chunk then
      REMOVE PP from PU
    else
      IDENTIFY the sequential chunk following PP
      PRODUCE an equivalent parallel chunk
      REPLACE the sequential chunk with the parallel chunk
      ADVANCE PP to the end of the new parallel chunk
      if an all-thread chunk was created then
        for all outer-conditionals in the chunk do
          INSERT a new parallelisation point into each clause of the conditional
        end for
        for all outer-loops in the chunk do
          INSERT a new parallelisation point into the loop body
        end for
        for all calls to user program units do
          INSERT a new parallelisation point into the start of the called program unit
          Add the program unit to the list of units to parallelise
        end for
      end if
    end if
  end while
  REMOVE PU from list of units to parallelise
end while
```

Figure 4. Parallelisation algorithm.

been at most two.

It should be noted that the above techniques assume that recursion is not used. While this is sound for Fortran 77, these techniques would have to be modified if the compilation method was to be adapted for use with other languages.

Caveat In Fortran 77, the `SAVE` declaration may be used to specify local variables that will be preserved between subsequent calls to their program unit. As the cloning scheme creates multiple independent versions of each program unit, this mechanism is prevented from working correctly. An implementation must ensure that variables named in `SAVE` declarations are shared correctly between clones.

2.2. Optimisation

The parallelisation method always classifies control statements (except for loops that are parallelised) as all-thread statements. If the entire body of a loop is a one-thread chunk, the method will generate a loop with a synchronisation at the beginning and end of the loop body, as depicted in Figure 6. The number of synchronisations caused by such loops can be large (especially if the loop is an inner loop), and may cause the code to execute inefficiently on many processors. It would be preferable in such cases to move the synchronisations outside of the loop; for n iterations, this would result in $2(n-1)$ less synchronisations.

When parallelising a loop, commands that alter control flow may be included in the cooperative chunk, as long as the chunk still has a single entry point and a single exit point. This scheme can be extended to one-thread chunks: a `DO` loop containing only a one-thread chunk may be removed from its all-thread chunk, and become a new one-thread chunk; `IF` statements can be treated similarly. The creation of new one-thread chunks may give rise to the situation where adjacent one-thread chunks exist; these can be coalesced. The process may then proceed in an outward sweep through the program unit. In some cases, a parallel clone may be reduced to a single one-thread chunk. In such circumstances, the parallel clone is redundant, and all calls to that clone should be replaced by one-thread calls to the sequential version of the program unit. To

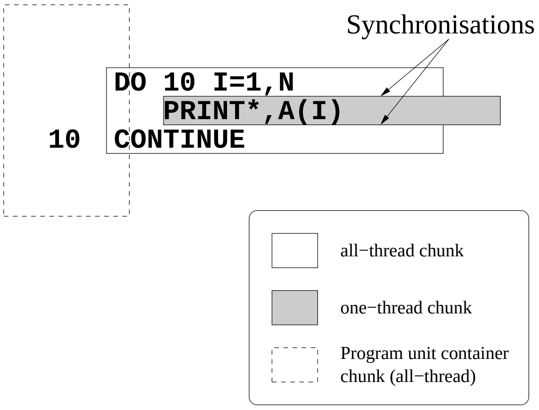

Figure 6. Synchronisation placement for loop with one-thread body.

maximise opportunities for reducing synchronisations, this process should begin with program units at the leaf nodes of the program call tree, and sweep outwards, finishing with the main program.

2.3. Summary

Due to space constraints, only a brief overview of this compilation method has been given here; a full description can be found in [10, Ch. 3].

3. Implementation

The compilation method described in the previous was implemented as an interactive compiler, MRes. Due to time constraints, only rudimentary dependence testing was included in the prototype; where testing is inconclusive, the user is consulted. Hence, the prototype is only suitable for use by an expert in parallelisation. Although MRes can generate parallel code for programs written in a large subset of ANSI Fortran 77 [1], no additional loop transformations are applied; these must be applied manually where required. Details of this implementation are given in [10, Ch. 4].

4. Evaluation

The main goal of this evaluation is to verify the Constant Plurality Hypothesis. To this end, several benchmark codes were parallelised using MRes; and the performance of these parallel codes measured; comparisons are made to hand-parallelised versions where these are available. A way of experimentally judging effective parallelisation for a parallel code is proposed in Section 4.1. In Section 4.2, the benchmark codes are introduced; in Section 4.3 a precise description of the evaluation environment is given. In Section 4.4, the evaluation procedure is explained, and the results are presented in Section 4.5. Supplementary information, too voluminous for inclusion in this section, appears in [10, App. B], and is reproduced electronically on the author's World Wide Web site.

4.1. Effective Parallelisation

Effective parallelisation implies that the performance of a parallel code gets close to some upper bound for that program. For a code to be effectively parallelised on a multiprocessor machine, all implicit coarse-grained parallelism should be exploited. Unfortunately, there is no known way to measure and quantify available implicit parallelism, in the general case. In the absence of a precise upper bound for performance, a pragmatic approach is required.

On a given quiet machine,[15] if the best known sequential version of a program runs in time T_S, then on p processors, no parallel version should be able to run faster than $\frac{T_S}{p}$. If *speedup* is defined for p processors as $S_p = \frac{T_S}{T_p}$ (where T_p is the execution time for the parallel code on p processors), this bound would be achieved where $S_p = p$; this is termed *linear speedup*. While linear speedup is an intuitive upper bound, running programs on multiple processors typically makes available larger amounts of fast cache memory. This means that in practice, the bound can be exceeded, resulting in *superlinear speedup* [6]. However, for many codes, linear speedup is not achievable, as some parts of the program cannot be parallelised. Therefore, linear speedup *can* be taken as a reasonable upper bound; a parallel code achieving approximately linear speedup should be considered to be effectively parallelised. Where this is not possible, a comparison with a hand-parallelised version of the program should be made; again, a favourable comparison implies effective parallelisation.

4.2. Test Codes

The benchmark codes used in the evaluation come from two sources: first from the SPEC CFP95 Benchmark Suite; and secondly from a local collection of codes that present interesting challenges for parallelisation. Each code is described, and assigned an identifier which will be used subsequently to refer to the code.

[15] A "quiet machine" is typically defined as a machine which is running no (other) user processes.

4.2.1. Local Benchmarks

The following benchmarks were either developed, or else extensively studied, by the Centre for Novel Computing.

SHALLOW is a numerical weather prediction program based on the shallow-water equations; a system of equations is solved using finite difference approximations on a rectangular grid. The code is a deliberately naïve implementation, especially with regard to memory locality issues, and is designed to test the ability of parallelising compilers on realistic codes. The benchmark computes a grid size of 900×900 over 100 iterations.

CLUSTER simulates the behaviour of magnetisable "grains" on a disk surface (i.e. a two-dimensional thin film) when an external, uniform magnetic field is applied. The magnetic fields of the grains are calculated, and this data is used to plot a hysteresis loop for the disk surface. The author has extensively studied the manual parallelisation of CLUSTER using different parallel programming paradigms, and on various architectures, including shared memory multiprocessors [9]. In this study, 3072 magnetisable grains are simulated over fifty steps, each of which requires a (variable) number of iterations of the ODE solver.

MOLDYN simulates the behaviour of a fluid's particles in a fixed space using molecular dynamics techniques. In particular, the forces governing the attractive and repulsive inter-particular forces are modelled by the commonly-used Lennard-Jones 6-12 potential energy function. The manual parallelisation of this benchmark has been studied extensively by Riley [13], who provides a more detailed description of the code. The benchmark currently simulates 4000 Argon atoms, held in a cube with periodic boundary conditions, over fifty time-steps.

SPECTRAL is a numerical weather prediction kernel, which was developed for use by the European Centre for Medium-range Weather Forecasts (ECMWF). In the spectral weather forecast model, calculations are performed in three spaces: physical space, Fourier space, and spectral space. This kernel repeatedly shifts dummy data from Fourier space to spectral space and back, using *Legendre Transformations*. The spectral space used by the benchmark is a grid of $8 \times 2 \times 5886$ points; fifty transformation iterations are performed.

TRED2 is a subroutine from the Eigensystem Subroutine Package (EISPACK) which reduces a real symmetric matrix to a symmetric tridiagonal matrix, accumulating the orthogonal transformations.[16] TRED2 contains imperfectly nested loops and several `GOTO` statements, and is hard to parallelise well; a comparison of manual and automatic parallelisation methods as applied to TRED2 is the subject of [12]. For the purposes of this evaluation, an input matrix of size 2048×2048 is used.

4.2.2. SPEC CFP95 Benchmarks

The SPEC CFP95 Benchmark Suite, i.e. the floating-point component of the popular CPU95 Benchmark Suite, is a collection of scientific Fortran 77 codes, designed to provide a portable measure of computers' abilities to compile and efficiently execute numerically intensive codes. Most of the codes display a large amount of coarse-grain parallelism, containing expensive loop nests iterating over large data structures.

TOMCATV is a vectorised mesh generation program, which decomposes an irregular computation domain into many finite volumes. TOMCATV solves, in a transformed space, two non-linear Poisson equations for the coordinates in the physical space. The code solves a 511 by 511 system of equations over 750 iterations.

SWIM, like SHALLOW, is a numerical weather prediction program based on the shallow-water equations. In this case, the program is derived from a code written by P. Swarztrauber, and has a completely different structure from SHALLOW. SWIM uses a grid size of 512×512 points (smaller than SHALLOW), but executes 900 iterations.

SU2COR is a quantum physics code which computes the glueball masses of elementary particles within the framework of Quark-Gluon theory, utilising a "warm bath" Monte-Carlo method. The supplied input files configure the benchmark code to use a four dimensional grid, size $16 \times 16 \times 16 \times 16$; forty measurement steps are performed (not preceded by any thermalisation steps), with results being output every five steps.

HYDRO2D is a program for simulating astrophysical jets. The behaviour of the jets is modelled using

[16] TRED2 differs from most of the other benchmark codes in that it is not a simulation running for a number of time-steps.

hydrodynamical Navier-Stokes equations. The benchmark uses a cylindrical coordinate system, with 402 points in the Z-direction and 160 points in the R-direction; 200 time-steps are performed.

MGRID is a simple multigrid solver, which computes a three dimensional potential field. The code solves a constant coefficient equation on a uniform cubical grid, and solves only a single equation, representing a scalar field. The benchmark has been adapted from the MG kernel in the NAS Parallel Benchmarks. The computation takes place over forty time-step iterations on a $64 \times 64 \times 64$ sized grid.

APPLU, which is based on the NAS Parallel Benchmark code SSOR, demonstrates the solution of a system of partial differential equations (PDEs) using the Symmetric Successive Over-Relaxation (SSOR) method from the field of computational fluid dynamics. In its current state, the benchmark simultaneously solves five different PDEs on a grid of size $32 \times 32 \times 32$ points for 300 iterations.

TURB3D simulates isotropic, homogeneous turbulence in a cube with periodic boundary conditions in all three coordinate directions. The Navier-Stokes equations are solved using a pseudo-spectral method, and a leapfrog-Crank-Nicolson scheme is used for time stepping. The simulation takes place over 111 iterations on a $64 \times 64 \times 64$ point grid.

APSI is a modification of the ADM code from the PERFECT Benchmark Suite; it simulates pollutant concentrations and deposition patterns in lake shore environments. The supplied input file runs the simulation on a $256 \times 1 \times 64$ point grid for 960 time-steps.

FPPPP is a quantum chemistry benchmark which computes two electron integral derivatives (an operation used in the popular "Gaussian" software package). The main loop of this code does not perform time-step iterations, but iterates over the electron shells of the atoms; 130 atoms are simulated.

WAVE5 is used to simulate plasma phenomena for a number of particles which are located on a two-dimensional cartesian mesh. This is achieved by solving Maxwell's equations and particle equations of motion. The benchmark distributes 750000 particles on a grid of size 1250×60 and executes forty time-steps.

4.2.3. Summary

The benchmark codes vary in many ways, including their size and complexity; Table 1 summarises various properties of the codes.

4.3. Execution Environment

The evaluation was performed on a 16 processor SGI Origin 2000; this machine is based on the cc-NUMA (Cache-Coherent Non-Uniform Memory Access) architecture, and provides cache-coherent shared memory at the hardware level, although physically, the machine's memory is distributed. The machine contained MIPS R10000 CPUs, clocked at 195 MHz. Each processor chip contains a 32 Kbyte, 2-way set-associative instruction cache, in 64-byte lines; and a 32 Kbyte, 2-way set-associative, 2-way interleaved data cache, in 32-byte lines. Each processor also has access to a 4 Mbyte, 2-way set associative, unified level two cache, divided into 128-byte lines, and clocked at 130MHz. Pairs of processors (and their caches) are located on node boards, which collectively contain the machine's main memory, totalling 6 Gbytes in this case. During the experiment, the machine was running IRIX 6.5 (extended release name 6.5.9f).

Compilers from the MIPSpro Compilers 7.3.1.1m distribution were used to compile the parallel programs produced by MRes, and to produce sequential versions of the codes. Unless otherwise stated, all codes were compiled with the following optimisation flags: `-n32 -mips4 -r10000 -Ofast=ip27 -LNO:opt=1 -OPT:IEEE_arithmetic=3`. SGI provide optimised mathematics routines and a fast version of a subset of the standard UNIX mathematics library; both of these are linked before the standard mathematics library by specifying "`-lcomplib.sgimath -lfastm -lm`" during all compilation unless otherwise stated.

4.4. Methodology

The benchmark code is first compiled and linked sequentially with `-O2`, and the standard mathematics library (specified by `-lm`). At this optimisation level, no restructuring of numerical operations occurs, so the code is guaranteed to produce the same output as unoptimised code. The output from running this code is retained for verifying the validity of the parallel and optimised sequential codes.

Next, a parallel code is developed using MRes. In order to separate limiting effects of fixed plurality from limitations of the implementation, a perfect implementation of the compilation method must be imagined;

Benchmark	*Static Properties*			*Dynamic Properties*	
	Program Units[a]	`DO` *Loops*	*Code Lines*[b]	*Problem Size*	*Time-step Iterations*
CLUSTER	20	171	3180	3072 grains	50
SHALLOW	1	85	351	900×900	100
MOLDYN	11	28	316	4000 particles	50
SPECTRAL	7	29	296	$8 \times 2 \times 5886$	50
TRED2	1	35	143	2048×2048	N/A
TOMCATV	1	16	106	511×511	750
SWIM	6	24	259	512×512	900
SU2COR	34 (+1)	117	1558	$16 \times 16 \times 16 \times 16$	40
HYDRO2D	42	163	1710	402×160	200
MGRID	12	46	330	$64 \times 64 \times 64$	40
APPLU	16	168	2423	$32 \times 32 \times 32$	300
TURB3D	23	70	1280	$64 \times 64 \times 64$	111
APSI	96	298	4225	$256 \times 1 \times 64$	960
FPPPP	38	39	2115	130 atoms	N/A
WAVE5	104 (+1)	362	6213	1250×60	40

[a] `ENTRY` points are counted as program units; `BLOCK DATA` declaration counts appear separately in parenthesis, where present.

[b] Code lines do not include comments or blank lines; continuation lines are included.

Table 1. Some static and dynamic properties of the benchmark codes.

one which can work beyond the limited language subset supported by MRes, and which can transform loop nests arbitrarily to maximise parallelism while retaining good locality. For each benchmark, a sequence of sequential versions is developed, in which the following changes are made:

- unsupported features of Fortran 77 are rewritten;
- loop nests are transformed where this uncovers parallelism and/or improves locality; and
- the storage of array dimensions are reordered where this improves locality.

No algorithmic changes are made.

As stated above, parallel codes that achieve close to linear speedup can be judged as being effectively parallelised, as can codes whose performance compares favourably with that of a hand-parallelised code. Codes not meeting these criteria may be examined further to explore whether effective parallelisation would be possible with constant plurality. Where the MRes parallel code performs significantly worse than other versions, the reasons for poor performance must be investigated, to determine whether or not effective parallelisation was prevented by constant plurality.

MRes parallel codes are then compiled with the options given above, as is each sequential code. As benchmark versions of codes typically run for an artificially short time,[17], if the execution time for the whole code is measured, the cost of initialisation and termination will be artificially significant. This is especially relevant when measuring speedup, as these parts of the code often perform I/O and contain little implicit parallelism. To prevent this adversely affecting results, initialisation and termination are excluded.

Execution times are measured when the machine is "quiet", i.e. when only the user running the experiments can log in. Despite these precautions, experimentation has shown that the evaluation machine does not produce reproducible results for 16 processors. As the evaluation is intended to measure the performance characteristics of the parallel codes, rather than the limitations of the execution environment, no measurements have been made using sixteen threads. Due to the complexity of the operating system, there will still be some sporadic operating system activity when the parallel codes are run. To avoid execution times being distorted by fluctuations in operating system activity, performance results for a given executable are obtained by measuring the core computation timer value over three consecutive runs and taking the mean.

[17] For example, CLUSTER performs less than 10% of the calculation the code was written to perform.

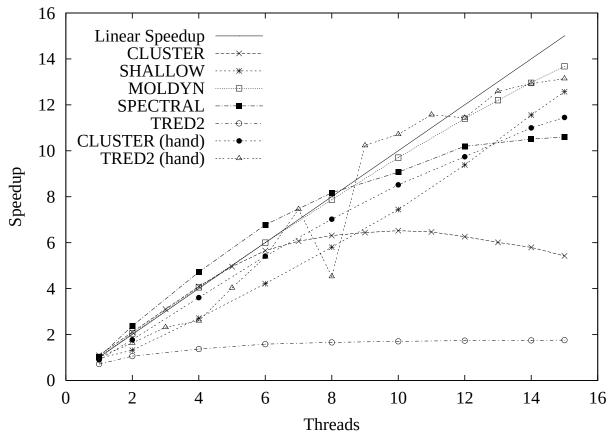

Figure 7. Speedup of local benchmarks.

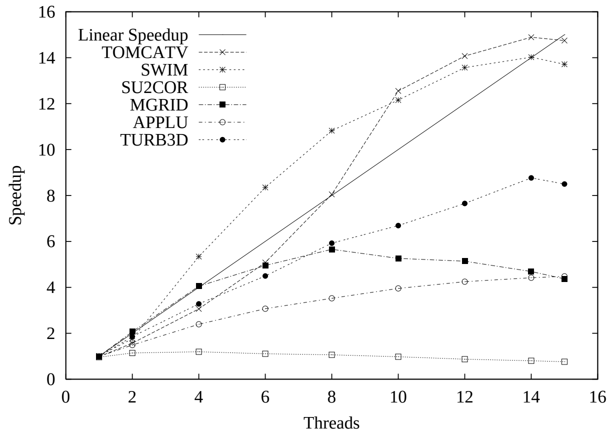

Figure 8. Speedup of SPEC CFP95 benchmarks.

4.5. Results

For each benchmark, the speedup of the fastest MRes code was calculated using the equation from Section 4.1, comparing performance to the fastest sequential version. The results for the local codes and the SPEC CFP95 benchmark suite are shown in the graphs in Figures 7 and 8, respectively; results are not shown for HYDRO2D, APSI, FPPPP and WAVE5, which could not be parallelised using MRes. The hand-parallelised code for CLUSTER was produced by the author during earlier work [9]; the development of the hand-parallelised code for TRED2 is described in [12].

Of the codes not parallelised by MRes, HYDRO2D, FPPPP and WAVE5 all use `EQUIVALENCE` and/or `COMMON` to overlay variables in memory. The other code, APSI, creates a single large array, `WORK`, and all arrays depending on the problem size are calculated as offsets into this array; this allows the problem size to be changed without recompilation. These codes all violate the assumption of the Constant Plurality Hypothesis, that each data structure must be a separate entity, capable of having its own plurality. While it would be possible to rewrite these codes to adhere to the assumption, this was not possible in the time available.

4.6. Discussion

The fastest MRes parallel codes for MOLDYN, SPECTRAL, TOMCATV, and SWIM achieve approximately linear speedup; these benchmarks have been effectively parallelised. An examination of the hand-parallelised version of TRED2 reveals that the plurality of each variable is fixed throughout execution. In the MRes parallel code for SHALLOW, every loop nest, except for the timestep loop, is parallelised. Therefore, TRED2 and SHALLOW can be effectively parallelised with constant plurality applied.

The codes for CLUSTER and MGRID start with linear speedup, but performance tails off with higher numbers of threads. During the timed section of CLUSTER, the MRes code performs 28078 synchronisations, whereas the hand parallelised version performs only 9120 synchronisations; the MRes code for MGRID performs 159627 synchronisations. In both these codes, the tail-off in performance is most likely due to synchronisation cost, rather than ineffective parallelisation. It is likely that if a fork/join code generation had been used, these codes would perform better for large numbers of processors.

As shown by the graph in Figure 8, the MRes code for TURB3D displays considerable speedup, although falls short of linear speedup. APPLU performs less well, but this is due to loop-carried dependence in two significant loops; through transformation, inner-loop parallelism is uncovered. It is doubtful whether there is any implicit parallelism in these codes that is not exploited in the final MRes codes; however, without a comparison to a hand-parallelised version, effective parallelisation cannot be verified. The MRes code for SU2COR shows no significant speedup. However, as with TURB3D and APPLU, no hand-parallelised code was available, and so it is not known how much implicit coarse-grained parallelism is present in the code.

5. Related Work

Various studies of privatisation (for scalars and arrays) have been made [3, 7, 8], but privatisation is always considered to be a loop transformation. For further background on loop transformations, the reader is referred to [2, Sec. 6].

The cloning scheme described in Section 2.1.1 is related to the cloning scheme of Cooper *et al.* [4], in that it creates different versions of a procedure for different call scenarios. However, while both schemes address problems regarding procedure boundaries, the scheme introduced by Cooper *et al.* is targeted at procedure boundaries inside loops that are being optimised.

6. Conclusions

In the evaluation described in Section 4, it was shown that six of the fifteen benchmark codes could be effectively parallelised, with all variables having constant pluralities, thus verifying the Constant Plurality Hypothesis to be correct. The hypothesis is based on the assumption that each data structure in the code has a single purpose, and can be assigned its own plurality, i.e. can be independently privatised; four benchmarks did not comply with this assumption (due to their use of `EQUIVALENCE` and `COMMON` declarations to overlap variables in memory).

It is tempting to make the conjecture that any code fitting these assumptions can be effectively parallelised with all variables having constant pluralities. The greatest difficulty in finding counter-examples to this conjecture is determining the amount of implicit coarse-grained parallelism contained in a code; e.g. in the case of SU2COR, it is not known how well the code *can* be parallelised. Finding a counter-example to the conjecture will require the use of codes which have been studied in depth, and for which a good hand-parallelised code is available. If such a code requires the variation of any data structure's plurality in order for implicit parallelism to be exploited, it will invalidate the conjecture.

References

[1] American National Standards Institute (ANSI), *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, American National Standards Institute, Inc., New York, 1978.

[2] D. F. Bacon, S. L. Graham, O. J. Sharp, "Compiler Transformations for High-Performance Computing", *ACM Computing Surveys*, **26**-4, Dec. 1994, pp. 345–420.

[3] W. Blume *et al.*, "Automatic Detection of Parallelism: A Grand Challenge for High-Performance Computing", *IEEE Parallel and Distributed Technology*, **2**-3, Fall 1994, pp. 37–47.

[4] K. D. Cooper, M. W. Hall, K. Kennedy, "Procedure Cloning", in *Proceedings of the 1992 International Conference on Computer Languages* (Oakland, California, Apr. 1992), IEEE Computer Society Press, 1992, pp. 96–105.

[5] R. Cytron, J. Lipkis, E. Schonberg, "A Compiler-Assisted Approach to SPMD Execution", in *Proceedings of Supercomputing '90*, IEEE Computer Society Press, pp. 398–406.

[6] D. P. Helmbold, C. E. McDowell, "Modeling Speedup(n) Greater than n", *IEEE Transactions on Parallel and Distributed Systems*, **1**-2, Apr. 1990, pp. 250–256.

[7] D. J. Kuck *et al.*, "Dependence Graphs and Compiler Optimizations", in *Proceedings of the 8th ACM Symposium on Principles of Programming Languages (POPL '81)* (Williamsburg, Virginia, Jan. 1981), ACM Press, 1981, pp. 207–218.

[8] Z. Li, "Array Privatization for Parallel Execution of Loops", in *Proceedings of the 1992 International Conference on Supercomputing* (Washington, D. C. July 1992), ACM Press, 1992, pp. 313–322.

[9] J. M. MacLaren, J. M. Bull, "Lessons Learned when Comparing Shared Memory and Message Passing Codes on Three Modern Parallel Architectures", in *Proceedings of High-Performance Computing and Networking (HPCNE'98)*, LNCS **1401**, Springer-Verlag, 1998, pp. 337–346.

[10] J. M. MacLaren, *Constant Plurality: A New Concept for Automated Parallelisation*, PhD Thesis (submitted), Department of Computer Science, University of Manchester, May 2001.

[11] N. Mukherjee, J. R. Gurd, "A Comparative Analysis of Four Parallelisation Schemes", in *Proceedings of the 1999 ACM International Conference on Supercomputing*, ACM Press, pp. 278–285.

[12] M. F. P. O'Boyle, J. M. Bull, "Expert Programmer versus Parallelizing Compiler: A Comparative Study of Two Approaches for Distributed Shared Memory", *Scientific Programming*, **5**-1, 1996, pp. 63–88.

[13] G. D. Riley, J. M. Bull, J. R. Gurd, "Performance Improvement Through Overhead Analysis: A Case Study in Molecular Dynamics", in *Proceedings of the 1997 International Conference on Supercomputing*, ACM Press, pp. 36–43.