

An Optimization Experiment with the Community Land Model on the Cray X1

James B. White III, *Center for Computational Sciences, Oak Ridge National Laboratory*

ABSTRACT: *Version 2.1 of the Community Land Model (CLM) uses data and control structures that challenge the capabilities of the Cray Fortran compiler. We describe an optimization experiment where we modified the CLM data structure within a computationally expensive tree of subroutines and compared performance with the original code. The modifications produce a 20% increase in performance on an IBM p690 and 5.86 to 7.29 times this performance on a Cray X1. The modified code may also be as maintainable and extensible as the original code.*

1. Introduction

The Community Land Model (CLM) is the land model for the Coupled Climate System Model (CCSM) and the Community Atmospheric Model (CAM) [Vertenstein]. CLM may also be run in stand-alone mode using atmospheric data sets. As part of a larger effort between NCAR, Cray, and Oak Ridge National Laboratory (ORNL) to optimize CCSM on the Cray X1, we are investigating the performance potential of CLM on the X1 at the Center for Computational Sciences (CCS) at ORNL.

CLM uses a horizontal grid matching that of the atmosphere in CAM, along with a subgrid for modeling multiple plant and surface conditions within a single atmospheric grid cell. Version 2.1 of CLM includes revised data structures designed to allow flexibility in the subgrid representation. These data structures and the resulting loop structure present significant barriers to vectorization.

To investigate strategies for improving the performance of CLM, we performed an experiment where we modified the data structures within a computationally intensive tree of subroutines and compared the performance with the original subroutines.

In Section 2, we describe the data structures and control flow of CLM 2.1, and we describe our experimental structures and implementation in the Sections 3 and 4. In Section 5, we describe results from single-process runs on the Cray X1 and IBM p690 in the CCS, and, in Section 6, we summarize the results and their potential impact on development plans for CLM.

2. CLM 2.1

The data structures in CLM 2.1 have grid cells at the highest level, where each grid cell represents a certain physical area of land on Earth [Vertenstein]. Below grid

cells are land units, columns, and plant functional types (PFTs). Each level can have multiple instances of each sublevel, where sublevel areas represent percentages of the parent area, not physically contiguous regions. Land units describe different soil properties and land-cover types, such as lakes and glaciers. Columns describe different soil states. Also, energy and particle fluxes with the atmosphere are computed at the column level. PFTs each represent broad categories of plants that may compete for the column resources.

CLM is written in Fortran with MPI and OpenMP parallelism, and each subgrid level is implemented as a user-defined type. Grid cells are independent and are distributed among processors. Within each type are many additional types for collections of related variables, such as physiological parameters, state variables, and flux variables for energy, momentum, and various chemicals. The type for a given level has an array pointer for the next sublevel and a pointer to its parent level. For example, each column has a pointer array of PFTs and a single pointer up to its parent land unit. These types are all defined in a single module, "clmtype.F90".

A second module, "clmpoint.F90", defines pointer arrays to one-dimensional aggregates of each sublevel. For example, "ppoint" contains a pointer to an array of all the PFTs on that processor. Each column type points to a PFT subarray within "ppoint", and each PFT type contains an integer that gives the index of that PFT within "ppoint".

Most of the runtime of CLM is within the time-stepping loop in the subroutine "driver". At each time step, an inner loop iterates over each column in the aggregate array of all columns on that processor. Each column is passed to a series of subroutines that then operate on that column and its PFTs. The subroutines read variables defined at the land-unit and grid-cell levels, and they access those variables by following a pointer in the column type. The driver calls

different subroutines depending on whether the column is part of a “lake” land unit or not.

Each subroutine called by “driver” declares local pointers for the variables used within the type hierarchy. The subroutine then proceeds with a long list of pointer associations, pointing down the type hierarchy directly to each variable. The remainder of the subroutine then uses the local pointers as shorter aliases for the “clmtype” variables.

3. Experiments

The results we describe are based on our experiences on Cheetah and Phoenix in the CCS. Cheetah is a cluster of 27 IBM p690 nodes, where each node has 32 Power4 processors running at 1.3 GHz. The Cheetah operating system is AIX 5.1, and the Fortran compiler is XL Fortran 7.1.1. Phoenix is a 32-MSP Cray X1 running at 800 MHz. The Phoenix operating system is UNICOS/mp 2.1, and the Fortran compiler is Cray Fortran 4.3.

The high-level outer loop and the complicated data structures in CLM 2.1 present a major challenge for the X1 compiler. The compiler is able to vectorize and multistream the original code only a trivial amount. ORNL is conducting two independent experiments to investigate strategies for optimizing CLM for the X1. In one we modify the data structures, and in the other we attempt to leave the data structures unmodified. We describe only the former experiment here.

The goal of the experiment is to test the following hypothesis: structures implemented strictly with modules and arrays of built-in types can provide similar flexibility to the original structures—but can provide much higher performance on vector systems and similar or better performance on superscalar systems. The experiment is to implement a tree of subroutines called from the “driver” loop and compare performance with the original tree. Data are copied from the original to the modified structures immediately before each call, and they are compared afterwards to ensure that differences are within appropriate numerical round-off errors. We chose the tree under the subroutine “Biogeophysics1”, which is the most computationally expensive tree according to a profile generated on the IBM p690.

4. Modifications

Our modified data structures have no user-defined types. The abstract types representing grid cells, land units, columns, and PFTs are implemented as Fortran modules. Within each module, each scalar variable is promoted to a one-dimensional array, and each array gets an additional leading dimension. The original sub-types within each type are thus “flattened” into separate variables. To correct redundant variable names created by this flattening, and to avoid indexing errors, each variable has an additional single character prefix: “g” for grid-cell variables, “l” for land-

unit, “c” for column, and “p” for PFT. Relationships between types are implemented as index arrays. For example, the integer array “pcolumn” gives the index of the column associated with each PFT.

The layout of the array variables follows conventions dictated by their use. Significant portions of the computations within “driver” are specific to land units that do not represent lakes, so the column and PFT variables are sorted such that the non-lake points are all first. This arrangement allows efficient specification of lake versus non-lake points through array bounds.

The PFT variables are further sorted to allow efficient reductions to column variables. CLM 2.1 uses a single PFT for each column by default, but the data structure and implementation are designed to allow multiple PFTs per column, as required by the planned addition of competition for resources among plant types [Hoffman]. Variables at the PFT level may be summed or averaged at the column level. To enable vectorization of such operations, the PFTs are sorted into blocks of independent columns. Thus, the first PFTs associated with each column is grouped, then the second, and so on. An index-bounds array, such as that used for compressed storage of sparse matrices, designates the extent of each independent-column block. Because each block of PFTs is associated with an independent set of columns, column updates may be vectorized and multistreamed over each block.

Using these new data structures, we modified “Biogeophysics1” and all subroutines it calls that are not inlined automatically by the compiler using default optimization and the “-Omodinline” option. The other subroutines requiring modification were “BareGroundFluxes”, “CanopyFluxes”, “FrictionVelocity”, and “SurfaceRadiation”. We also modified the function “StabilityFunc”, used by “FrictionVelocity”, but these modifications were scalar strength-reduction and input-specialization optimizations, not data-structure modifications.

Calls to “Biogeophysics1” in the original code occur within an “if” test for non-lake columns, all within a loop over columns. Each “Biogeophysics1” call passes a single column as an argument. We originally replaced the column loop with a single “Biogeophysics1” call that passes bounds to the column arrays and PFT arrays, bounds that restrict operation to the non-lake columns. By passing bounds, however, we were also able to introduce a simple but effective tuning parameter. We added a new outer loop with a large stride and used the loop index and stride to define array blocks. The stride, and thus the size of the resulting blocks, is tunable for different systems. Small blocks can be used for cache-dependent superscalar systems, full-sized blocks for vector-only systems, or large blocks for vector systems with additional dimensions of parallelization, such as threads or streams.

Instead of passing complicated user-defined types as arguments, we “pass” the data through modules via “use” statements. The outer loop over columns moves into the

subroutines, and loops over PFTs for each column become larger loops over all the PFTs within the bounds now provided as arguments. The modified “Biogeophysics1” thus starts with a column loop. This loop reads variables defined at the land-unit and grid-cell level, so each column iteration must determine the appropriate index at the higher levels from a corresponding index array. The index for each level has the corresponding prefix, as shown in the following code fragment.

```
do ci = clb, cub
  li = clandunit(ci)
  gi = cgridcell(ci)
  ...
```

The appropriate index is then used for each variable. The following example uses grid-cell (“g”) variables to compute a column (“c”) variable.

```
cthm(ci) = gforc_t(gi) +
0.0098*gforc_hgt_t(gi)
```

The use of arrays instead of user-defined types introduces a source of errors because the wrong index can be used with a given variable. For example, the land-unit index “li” could be used with a column variable. Because there are more columns than land units, the index would never be out of range, so the resulting error could be difficult to detect. An important motivation for adding a prefix to each variable was to reduce the likelihood of such errors; the prefix of the index and of the variable must match. Note that this is strictly a convention and is not enforced by the compiler.

After the initial column loop, “Biogeophysics1” calls “SurfaceRadiation” and performs a PFT loop. Within the PFT loop, it calls either “BareGroundFluxes” or “CanopyFluxes” depending on the nature of the PFT. We modified “SurfaceRadiation” to take PFT bounds as arguments, and we expanded the PFT loop over the same bounds. We pulled the calls to “BareGroundFluxes” and “CanopyFluxes” out of the loop and passed PFT bounds as arguments. The “if” test to determine which flux subroutine to call was then moved inside each subroutine.

The original “SurfaceRadiation” subroutine is dominated by a PFT loop, so it required little structural change. It has an imperfectly-nested inner loop over radiation wavebands, of which there are only two. The compiler required the following directive on this inner loop to vectorize the outer loop.

```
!dir$ unroll(nband)
```

The “unroll” directive alone, without specification of “nband”, was not sufficient to induce vectorization.

The subroutine “BareGroundFluxes”, with the “if” test for bare ground moved inside it, introduced a new implementation issue: how to implement conditions around large blocks of code. Using a standard “if” statement

would result in a large number of masked vector operations, and thus many redundant computations. Instead we chose to implement an index filter. The following code illustrates the filter idiom for PFTs, where “<test>” represents the condition.

```
fn = 0
do pi = plb, pub
  if (<test>) then
    fn = fn+1
    filterp(fn) = pi
  end if
end do
```

The Cray compiler recognizes this idiom and produces vectorized code, though it does not yet produce multistreamed code.

The PFT loops then iterate over the “fn” values of “filterp”, and the PFT index comes from “filterp” instead of directly from the loop bounds. Note the use of “f” as the prefix for filter-related variables. The filter idiom fits naturally into the loop structure where index values come from index arrays. The following code fragment illustrates such loop structure, where the loop body uses PFT, column, and grid-cell variables.

```
do fi = 1, fn
  pi = filterp(fi)
  ci = pcolumn(pi)
  gi = pgridcell(pi)
  ...
```

A difference between the filter array and the other index arrays in this example is that the filter provides index values to be used on the left-hand side of assignment operations. The other indices are used only for reading, so their values may repeat. The following Cray directive instructs the compiler that the variable “filterp” is a permutation; none of its values repeat.

```
!dir$ permutation(filterp)
```

This directive is placed only at the point where “filterp” is declared. With this directive, the Cray compiler vectorizes loops using indices derived from “filterp”. We found that the compiler does not always multistream such loops, however, but does multistream the loops with the addition of a “concurrent” directive.

“BareGroundFluxes” calls the subroutines “MoninObukIni” and “FrictionVelocity”, each of which takes a list of scalar arguments. The Cray compiler inlines “MoninObukIni” but not “FrictionVelocity”, possibly because “FrictionVelocity” contains a string of “if-else-else” statements. Also, “FrictionVelocity” is called inside an iteration loop, though this loop has a static bound of three. We chose to pull the filtered PFT loop inside of the “FrictionVelocity” call, and this decision had a number of implications. The resulting splits of the PFT loop within “BareGoundFluxes” required many scalar temporaries to

be promoted to arrays. We used automatic arrays defined by the bounds provided as arguments, a strategy that can cause problems for systems with small limits on the program stack. We experienced no such problems on the X1 and p690, however.

Of all the subroutines we modified, we changed the structure of “FrictionVelocity” the most. The original subroutine takes scalar arguments and computes new values for three of those arguments. Each computation uses a four-level test of the following form, where “zetac” is a constant and “v” is the value to be computed. Computations of “v” can be computationally expensive, with fractional powers and logarithms.

```

if (zeta < -zetac) then
  v = ...
else if (zeta < 0.) then
  v = ...
else if (zeta <= 1.) then
  v = ...
else
  v = ...
endif

```

The modified “FrictionVelocity” takes array arguments, along with array bounds and a filter array. The filter argument provides the filter used by the caller. For each output array, the above “if” structure is replaced by the construction of four local filters. In the following example, “filterp” is the filter passed in, and “flnz”, “fl0”, “fle1”, and “felse” are local filters.

```

flnzn = 0
fl0n = 0
fle1n = 0
felsen = 0
do fi = 1, fn
  pi = filterp(fi)
  if (pzeta(pi) < -zetac) then
    flnzn = flnzn + 1
    flnz(flnzn) = pi
  else if (pzeta(pi) < 0.) then
    fl0n = fl0n + 1
    fl0(fl0n) = pi
  else if (pzeta(pi) <= 1.) then
    fle1n = fle1n + 1
    fle1(fle1n) = pi
  else
    felsen = felsen + 1
    felse(felsen) = pi
  end if
end do

```

Like for a single filter, this multi-filter construction is vectorized but not multistreamed by the Cray compiler. Separate computation loops then iterate over the elements of each filter, allowing for full vectorization and multistreaming of the expensive power and logarithmic computations.

In the original CLM code, all subroutines exist within modules, even if the subroutine is the only member of the module. We extracted each modified subroutine from its module, thus using modules exclusively for data “objects”.

“StabilityFunc” was unique in that it became a “contained” function of “FrictionVelocity”.

The arguments passed to the modified “FrictionVelocity” include arrays local to the caller, which have the same bounds as are passed to “FrictionVelocity”, and arrays defined in the PFT module, which have bounds beyond those passed to “FrictionVelocity”. With “FrictionVelocity” extracted from its module, we use the rules for argument passing that apply when no interface block is available. We pass caller-local arrays directly but pass PFT-module arrays using the element at the lower bound, as in the following example. The array bounds are “plb” and “pub”, arrays on the third and fourth lines are from the PFT module, and arrays on the last line are local to the caller.

```

call frictionvelocityv(
  plb, pub, fn, filterp, &
  pdispla(plb), pz0mv(plb), &
  pz0hv(plb), pz0qv(plb), &
  pobu, pum, pustar, ptemp1, ptemp2)

```

If “FrictionVelocity” were a module subroutine, we would pass subsections of the module arrays. Historically, Fortran compilers have done limited analysis of subsection arguments, often choosing the conservative approach of making copies. These unnecessary copies can have dramatic negative performance effects. We avoided the issue here, although the current Cray and IBM compilers may well have eliminated the unnecessary copies.

The “FrictionVelocity” subroutine is called by both “BareGroundFluxes” and “CanopyFluxes”. The various arguments cannot be “passed” by module because the two callers pass different arguments. Also, where “BareGroundFluxes” calls “FrictionVelocity” from an iteration loop with static bounds, “CanopyFluxes” calls it from an iteration loop with runtime bounds that are specific to each PFT.

This loop poses a new challenge for vectorization, but our solution uses a familiar concept, the filter array. We replaced the original test in the “do while” with a length test of the filter array; the iteration ends when the filter has no more elements. After a copy of the initial filter is made, the filter array is “weeded” at each iteration using the original test of the iteration loop. The initial filter array is then restored after the loop.

This and most other modifications were intended primarily to enable vectorization and multistreaming on the X1. We expected many of these modifications to also benefit performance on superscalar machines, such as the IBM p690, while other modifications might hamper performance.

The original CLM code is near-optimal in temporal locality of data, as all computations for each column are performed before using the next column. The original code may be weak in spatial locality, however, because variables are scattered throughout the hierarchy of user-defined types. We saw only limited correlation between the usage pattern

of variables and their spatial proximity within the user-defined types.

The modified data and code structures provide more spatial locality, with smaller loop bodies using values arranged contiguously in memory. Such smaller loop bodies and array variables—instead of large outer loops with nested-pointer variables—are also much friendlier to data-dependence analysis, so superscalar compilers have greater opportunities for pipelining and instruction-scheduling optimizations. The smaller loop bodies limit temporal locality, however, particularly with large loop bounds, where the cache may be refilled before data can be re-used. Still, the bounds are tuning parameters, so they can be reduced to improve temporal locality.

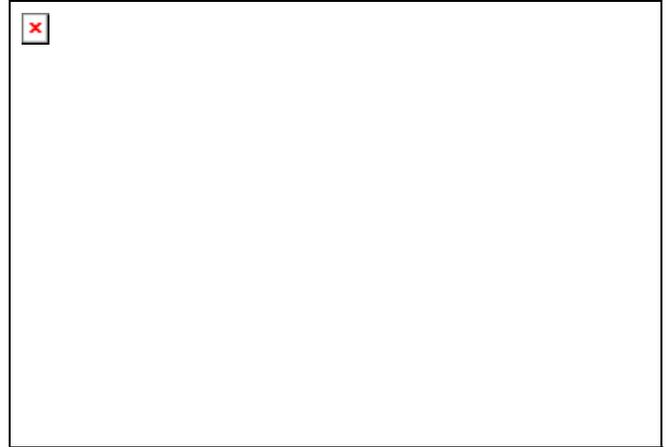
The various index filters, however, have no apparent benefit for superscalar systems, though they are essential for vectorization. They reduce spatial locality by fragmenting memory use, and they add the overhead of filter construction and index manipulation.

We conducted this experiment to determine the actual performance resulting from these various positive and negative factors.

5. Results

We ran CLM with the original and modified implementation of the “Biogeophysics1” tree for 48 time steps using the standard input files provided with CLM [CLM]. Runtimes for each implementation of “Biogeophysics1” came from the timer facility built into CLM. We ran on a single processor of the IBM p690 and a single MSP of the Cray X1. The results shown here represent tuned values for the block size on each system. On the p690, the best block size was 16 words, which happens to coincide with the 128-byte cache-line size. On the X1, the best block size was one fourth of the total array size. Splitting the bounds into four blocks allowed multistreaming at the highest level, over calls to “Biogeophysics1”. The call tree then needed no internal multistreaming.

The runtime results appear in the following figure. On the p690, the modifications give a net improvement of 20%. On the X1, the improvement is dramatic, some 98%. The modified code on the X1 runs 5.86 times faster than the modified code on the p690 and 7.29 times faster than the original code on the p690. Therefore, the modifications did indeed provide significant speed on the X1 without reducing performance on the p690.



The question of maintainability and extensibility of the resulting code is more difficult to judge, however. The modified code does allow some errors not likely to occur in the original. We have found that the matching-prefixes convention is effective at avoiding the use of the wrong index, but nothing at the compiler level enforces this convention. Also, one may forget to assign the index for a higher level before using it. In a PFT loop, for example, one may use the column index, “ci”, without first setting its value from “pcolumn(pi)”.

The original code has its own sources of error, however. Variables at different levels of the data structures—at the PFT level and column level, for example—may have the same name. These names are distinct in the context of their data types. In some CLM subroutines, however, the same local pointer is used for the variable at one level in one section of code and at another level in another section. Errors in the association of such a pointer could produce subtle failures that are difficult to detect and correct.

The use of local pointers has additional adverse effects, particularly for readability. Each subroutine starts with a long list of pointer declarations and associations that obscure the actual computation. The associations typically use the results of previous associations, making it difficult to determine the actual variable referenced by any given local pointer. The prefix convention we applied to the modified code eliminates much of this difficulty; it is clear from the name of a variable what level it comes from. We considered adding to the convention a mechanism for distinguishing local variables, but we implemented no such mechanism.

To help investigate relative maintainability, it may be useful to compare the size of the original code versus the modified code. The table at the end shows the lines of code and number of characters in each subroutine, with comments and blanks removed. Excluding “FrictionVelocity”, the modified subroutines have 29-56% fewer lines and 13-53% fewer characters. Even including “FrictionVelocity”, the modified subroutines

have a total of 23% fewer lines and 18% fewer characters than the originals.

Consider a modification to a subroutine that requires the use of an additional PFT variable, one that resides in the original hierarchical data structure, and thus in the PFT module of the modified code. In the modified subroutines, you simply use the variable directly. In the original subroutines, you must declare a local pointer and associate it. You must therefore modify three times as many lines of code in the original CLM than in the modified code. Because of examples like this, along with potential improvements in readability and reduced maintenance load, we believe that the modified data structures and coding conventions could prove at least as easy to maintain and extend as the original ones.

6. Conclusions

We have described data structures and coding conventions used to modify the tree of calls under “Biogeophysics1” within the main time-stepping loop of CLM. Our modifications improve performance on the IBM p690 and provide much higher absolute performance on the Cray X1. The resulting code appears to be similar to the original code in terms of maintainability and extensibility, perhaps even better. These results may prove adequate to justify a full conversion of CLM, and we are working with CLM developers to investigate this possibility.

Forrest Hoffman of ORNL is testing the strategy of modifying the CLM code to enable vectorization and multistreaming without modifying the original data structure. His experiments necessarily push the limits of the compiler more than those described here, so progress has been slower. The pace and success of his complementary experiments will factor heavily into decisions regarding the conversion of CLM.

	Lines			Characters		
	Original	Modified	Ratio	Original	Modified	Ratio
Biogeophysics1	225	98	0.44	4783	2234	0.47
SurfaceRadiation	119	61	0.51	2527	1344	0.53
BareGroundFluxes	205	138	0.67	4512	3122	0.69
CanopyFluxes	505	360	0.71	11179	9689	0.87
FrictionVelocity	120	248	2.07	2862	4943	1.73
Total	1174	905	0.77	25863	21332	0.82

Acknowledgements

The author thanks Forrest Hoffman, John Drake, and Pat Worley of ORNL, Mariana Vertenstein of NCAR, and Nathan Wichman of Cray for helpful discussions regarding the structure of CLM, future development plans, and optimization strategies.

This research was sponsored by the Mathematical, Information, and Computational Sciences Division, Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

About the Author

James B. White III, a.k.a. Trey White, is in the Scientific Application Support Group within the CCS. Trey is the primary CCS liaison for climate researchers funded by the DOE. CUG 2004 will be hosted by the CCS, and Trey is the Local Arrangements Chair. He can be reached at “whitejbiii@ornl.gov”.

References

- [CLM] Available from
[“http://www.cgd.ucar.edu/tss/clm/distribution/clm2.1/index.html”](http://www.cgd.ucar.edu/tss/clm/distribution/clm2.1/index.html).
- [Hoffman] Forrest Hoffman, personal communication.
- [Vertenstein] Mariana Vertenstein, Keith Oleson, Sam Levis, and Peter Thornton. *CLM2.1 User's Guide*. National Center for Atmospheric Research, January 2003. Available from
[“http://www.cgd.ucar.edu/tss/clm/”](http://www.cgd.ucar.edu/tss/clm/).