Extreme Vectorization in RELAP5-3D

Dr. George Mesina**, and **Peter Cebull
Idaho National Engineering and Environmental Laboratory

***ABSTRACT:**. Most of the work on vectorizing legacy Fortran programs was done in the 1980's and early 1990's. Developments by Cray, Inc. have made it possible to vectorize loops that could not be vectorized in those days. Therefore, legacy programs should be re-examined for possible vector speed gains. The remaining non-vector loops, subroutines, and programs are most challenging to vectorize. Among the most challenging, is the RELAP5-3D program that performs nuclear power plant modeling calculations for safety and simulator applications. Two subroutines were vectorized to increase overall code run speed by up to 33% on some problems. These two subroutines effectively had 4900 lines and 7100 lines of executable Fortran statements in single vectorizable do loops. This is extreme vectorization. Many of the techniques used to vectorize these subroutines are reported here.*

***Keywords:** RELAP5-3D, Cray, Vectorization, SV1, Nuclear power plant*

1. Introduction

RELAP5-3D is used to model nuclear power plants and predict the behaviour of these plants under a wide variety of operational and accident conditions. Worldwide, RELAP5-3D and its predecessor versions are used more often for nuclear power plant safety analyses than any other power plant modelling code. It has been adapted to nuclear power plant simulators and is used in training power plant operators. It is also seeing application in the design stages of both generation 3 and generation 4 power plants.

RELAP5-3D modelling features include multi-dimensional one- and two-phase flow, multi-dimensional heat transfer, multi-dimensional neutron kinetics, complete trips and control systems, and specialized models for specific plant components such as many different types of pumps and valves. Because of these features, it can be used in non-nuclear application areas. Current application areas include: nuclear fusion plant analysis, steam supply systems analysis, paper and pulp plant simulators.

This code has been under continuous development since the 1970s. It continues to grow with the changing requirements for nuclear power plant modelling and analysis and the improving capabilities provided by the computing industry.

In the mid-1980s through the early 1990s, an effort was undertaken to vectorize RELAP5-3D. The work was done mostly on Cray XMP and YMP platforms. The computing industry changed directions to various kinds of parallel processing and use of fast scalar chip beginning in the early 1990s; so code optimization efforts shifted to those and other trends in computing.

In recent times, there has been a renewed interest in vector computing. The INEEL has acquired a CRAY SV1 with a 3333-picosecond clock. With this machine available, the INEEL has undertaken an effort to optimize its application software for vector and vector-parallel processing; this effort includes RELAP5-3D.

It is noteworthy that Cray, Inc. has advanced the state of the art in vectorization since the days of the YMP. It is now possible to vectorize loops that contain inner loops. With this and other improvements, it is now possible to vectorize significant loops that could not previously be vectorized.

2. Performance Analysis of RELAP5-3D

RELAP5-3D/Version 2.2.4 was examined with Cray operating system performance measures. Perftrace 90.4 was run as RELAP5-3D executed four different calculations to determine where the performance enhancement could best be obtained.

The first input model is designated TYPPWR. It is a small input model having 139 control volumes and 142 junctions, which are connections between volumes. Most vector loops in the code are over subsets of volumes and junctions. Vector lengths are short in this problem.

The second problem is a model of the ROSA test facility used in performing design experiments for Generation 3 nuclear power plants. It has 448 control volumes and 469 junctions. It is and average-size problem.

The third input model is designated AP600. It is a full-scale model of the original Westinghouse 600 MW Generation 3 reactor. It has 1232 control volumes and 2230 junctions. This is a normal sized problem and has vector lengths averaging in the 50's.

The fourth model is designated 3DFLOW15. It is a rectangular pipe modeled with the full 3D capability of RELAP5-3D. It has 693 control volumes and 1728 junctions, but produces such large linear systems that most of the computation time is spent in the vectorized solver.

Among the subroutines listed as taking the most computational time in all four problems, were PHANTJ and PHANTV. They ranked first and second for the smaller problems but always in the top 5. A third subroutine, FORCES, whose output is seldom used and is not used in any of our test problems, ranked third in the small test cases. It was decided to make it active only through input when needed. Together, these 3 routines account for over 33% of the code run time in the two smaller problems. Theoretically, if vectorization (and shutting off) could reduce their runtime to zero, an overall code speed-up of approximately 50% could be achieved. This is summarized in Table 2.1

	PHANTV	***PHANTJ***	***FORCES***	***Total***
Test Case	Exec %	Exec %	Exec %	Exec %
TYPPWR	11.2	12.8	9.2	33.2
ROSA	11.6	13.0	9.3	33.9

Table 2.1 Percentage of run time spent in subroutines

Thus, it was decided to undertake the task to vectorize both PHANTJ and PHANTV to improve code performance with the goal of a 50% improvement in run speed on the INEEL Cray SV1.

3. Subroutine Structure and Effective Size

Neither PHANTJ nor PHANTV had ever been vectorized when the early vectorization work on RELAP5-3D was performed. Upon further examination, it was discovered that it had not been possible at the time, primarily because a loop could not be vectorized if it contained an inner loop. Both contain loops that have one or more inner loops.

3.1 PHANTV & PHANTJ Structure

PHANTJ is comprised mainly of one huge loop, DO 10, that is 1423 lines long. PHANTV is comprised mostly of two loops, DO 11 and DO 111, of lengths 1668 and 733 respectively, but it also contains about 30 other small loops. Most of the small loops vectorize naturally without any rewrite or directives, but some inherently cannot vectorize.

3.2 PHANTV & PHANTJ Size

There are six subprograms called from within the DO 11 loop of PHANTV, not counting intrinsic functions. One of these in turn calls subroutines. Counting all calls and uses of functions, there are 18 calls to subprograms within the loop. If these subroutines were to be replaced by their actual lines of executable code, they would add 2196 lines to its 1668 lines. Additionally, there is a large backward go to that often causes 1120 lines to be calculated a second time. The loop effectively has 4984 executable lines of code. A similar calculation shows that the DO 111 loop would expand out to 933 executable lines of coding.

There are five subprograms called from within the DO 10 loop of PHANTJ. Some of these in turn call subprograms so that 73 total subprogram calls are made from within the loop. If these subroutines were to be replaced by their actual lines of executable code, they would add 5088 lines to its 1433 lines. Again a large backward go to adds 660 lines. The loop effectively has 7171 lines.

Hereafter, the DO 10, DO 11, and DO 111 loops are referred to as the huge loops.

3.3 PHANTV & PHANTJ Speed

Both PHANTV and PHANTJ run at scalar speed on the Cray SV1. For some problems, PHANTV picks up some vector performance from the small vector loops, but since most of the work is in the huge loop that does not vectorize, the performance is basically at scalar speed. See Table 3.3.1 where Exe% is the percentage of the run time spent in the subroutine, MF means MFLOPS and Sec/C means seconds per call to the subroutine. Note that PHANTJ has lower MFLOPS rate and takes longer than PHANTV. Also, it runs slower as the size of the problem increases.

	PHANTV			***PHANTJ***		
Test Case	Exe%	MF	Sec/C	Exe%	MF	Sec/C
TYPPWR	9.2	13.5	.00268	12.8	10.9	.00305
ROSA	11.6	18.8	.0132	13.0	10.6	.0149
AP600	3.7	14.9	.0654	7.9	10.3	.139
3Dflow15	1.2	16.0	.0482	2.6	9.7	.103

Table 3.3.1. Speed measures

4. Vectorization Techniques

In order to vectorize the loop, the coding constructs that inhibit vectorization must be identified and overcome. The following is a list of some of the vector inhibitors in the huge loops:

- Subprogram calls
- Improper use of modules
- Variable length interior loops
- Backward GO TO
- Actual recurrence
- Apparent recursion
- If-tests too deeply nested

4.1 Subroutines and Modules

Subprograms are handled by inlining. For small subprograms, it is sufficient to list them on the compiler inline flag. However, for “large” subprograms (about 400 lines or more), a source code pre-compiler directive, namely INLINEALWAYS, must be added to force the compiler to inline it.

Inlining can introduce vector inhibitors to the loop that must be handled by rewriting the subprogram. Besides those vector inhibitors listed in Section 4.1, two other inhibitors resulting from inlining were the introduction of active output statements and mismatch of call arguments.

Call argument mismatch requires either the reworking of the variables declarations or the placing of passed data into variables of appropriate type, or both. In the case of RELAP5-3D, there were two problems with the arguments of subroutine POLAT.

First, an integer datum was passed in a floating-point array element and was declared an integer in POLAT. This arises from an old programming trick for saving memory. It equivalences all memory in RELAP5-3D to a single huge array, called FA, that is declared to have a large size, but which is reduced to the exact amount needed during execution via an operating system call. In the days when computer centers charged for supercomputer memory usage, this was done to reduce computer charges. All integer, floating point, and logical arrays were equivalences to FA. For machines with 32-bit integers and 64-bit (double precision) floats, in order to equivalence integers to floats, vectors of integers were promoted to matrices with an additional dimension (the first) being two. Thus pairs of integers were equivalenced to single floats. The second of a pair of integers carried actual data, while the other had junk.

One difficulty with this approach is in subprogram calling. All calculations and assignments of integer array entries, such as volno(iv), are promoted via a preprocessor to volno(2,iv). Floating point array elements are passed by simply placing the array element in the call sequence. With integers however, passing the element requires that the first index be one not two, e.g. volno(1,iv). In some places in the code, integer quantities were passed via a float, to which they were equivalenced, to avoid the call sequence complication. Compilers ignore this mismatch of data being declared floating point in the calling program but integer in the subprogram unless either an interface is used or the subroutine is inlined. Because POLAT is being inlined, this became an issue.

The second problem involved the fact that a scalar real was passed to POLAT in a position that was declared a real array. In fact, only a single datum was calculated and returned by the particular call. Once again, compilers allow this except for interfaces and inlining.

The first problem was overcome by using the actual integer array element. The second was solved by creating a one-dimensional, one-entry real array, vdummy. The scalar was copied into the array entry, passed to POLAT, and copied back into the scalar. Eventually however, POLAT was rewritten in structured Fortran 90 as LINT1; it passes a real scalar rather than an array of length one.

Modules will inhibit inlining unless certain rules are followed. First, the module itself must be compiled with a flag that says it is allowed to be inlined. The compiler documentation is written in such a way that one could read that it means the program using the module must specify that the module is inlinable. Second, use of allocatable arrays in inlinable modules is not allowed. One way to overcome this is by removing the contained subprogram that is to be inlined from the module and placing it in a separate file.

4.2 Code-length inhibitors

As noted, after the subroutines were inlined, the effective size of the DO 11 loop was nearly 5000 lines. The effective size of the DO 10 in PHANTJ was 7100 lines. These huge loops were too large for the compiler to vectorize. The compiler apparently runs out of internal storage for the analysis.

Use of compiler directives, such as IVDEP and CONCURRENT, cause the compiler to perform its operations while ignoring certain kinds of analyses. This sometimes allows the compiler to vectorize longer loops. Alone, neither directive caused the loop to vectorize. However, combining CONCURRENT with the aggressive compiler optimzation flag was sufficient to achieve vectorization, but only when all other inhibitors were commented out.

The strategy for vectorizing these subroutines was to mark coding that inhibited vectorization with a CPP pre-compiler flags. One kind of vector inhibitor was reworked at a time until it was overcome and the compiler could vectorize the loop with the alternate coding or appropriate technique. In some cases, the reason that a section of coding inhibited vectorization remained unknown until work on the section was underway.

It should be noted that aggressive compilation takes much longer than normal compilation. Now that all the vector inhibitors have been eliminated, it takes 500 seconds to compile PHANTV and 700 seconds to compile PHANTJ on the Cray SV1.

4.3 Inner loops

A loop can vectorize if it contains an inner loop, but the inner loop must both itself vectorize and have a fixed length. Simple things like a table lookup can break vectorization because the number of passes through the loop is variable. In PHANTV and PHANTJ, interpolation on a general table, preceded by a search in the table, and looping over the junctions that are attached to a control volume are important and occur more than once in the huge loops.

Some of these inhibitors were overcome by moving the calculations outside the huge loops. The results are stored in temporary arrays and used in the huge loops where needed. This was only possible because none of these calculations relied on any calculations done previously in the huge loop.

The table search for the critical Katatladze number, which was in an inlined subroutine, did rely on calculations from earlier in the huge loop. The variable length search through the table was replaced by a direct calculation. The table had a staggered mesh that caused the need for a search. A new table with a uniform mesh was constructed to include each of the staggered mesh grid points. A simple

formula for directly calculating the subinterval from the abscissa value was programmed. In this way the variable length search loop was eliminated by a direct calculation.

4.4 Recurrences

Worse than variable length inner loops are recurrences. These occur when a quantity is calculated in one iteration of the loop with a quantity calculated in a previous iteration. Examples include summing the entries of an array and bisection. Both of these occurred in inner loops within the huge loops of PHANTV and PHANTJ.

In the case of the sum loop, it did not rely on calculations done before it in the huge loop. It was handled in the same way as variable-length inner loops of this sort, by moving it before the huge loop. It was placed inside a loop with the same loop index and loop limits as the huge loop of PHANTV. The value of each sum was saved in a temporary array and used later in the huge loop.

The case of bisection was more complicated. The value of the Burington angle, b, is the solution of the non-linear equation:

$$b - \sin(b) = 2\pi\alpha_g, \qquad (4.4.1)$$

where α_g is the void fraction of the gas and b is measured in radians. This function has no analytical inverse and so a function, HTHETA, was written. It used a fifth order Taylor polynomial to approximate sine and calculated b via bisection as the zero of the function

$$f(b; \alpha_g) = 2\pi\alpha_g - b + \sin(b) \qquad (4.4.2)$$

The bisection was programmed to take 16 or 17 iterations depending on the value of void fraction.

Since bisection is inherently recursive, it had to be replaced in order to achieve vectorization.

A cubic spline interpolation to the inverse function was constructed. The values of the inverse function were calculated by using bisection to find the value of b for which

$$f(b; \alpha_g) = 0 \qquad (4.4.3)$$

for 200 different values of α_g. Chebyschev points were used to generate the values of α_g. This created a table of values that could be searched for the subinterval that contained any value of void fraction and then cubic spline interpolation could be applied to produce a good approximation to b. The table is constructed once and saved.

Normally, a search of this non-uniform mesh for a the containing subinterval would be required and that would be the kind of vector inhibitor discussed in Section 4.3, an inner loop of variable length. However, the formula for generating Chebyschev points is completely invertible and leads to a direct formula for calculating the subinterval that contains any void fraction. This is similar to the critical Katataladze number.

Overall, this new method of calculating Burington angle is faster than the original even on a non-vector machine because it reduces the number of operations by an order of magnitude.

4.5 False Recurrences

RELAP5-3D equivalences all arrays to a single large array. Because of this equivalence, the compiler interprets all array references as potential recurrences. This is solved by placing either the IVDEP or CONCURRENT compiler directive before each loop.

The compiler is programmed with unit stride through memory in mind. RELAP5-3D is programmed with a large stride that depends on the data type. For example, normally control volume data has a stride of 279 and junction data has a stride of 111. Unit stride is found in very few places.

In the control volume data block, data for the x-, y-, and z-direction are stored contiguously in data. For example, the wall friction of the gas, FWALG, can occur in all three directions Because everything is equivalenced to one large array, the wall friction in the x-, y-, and z-directions for volume IV are indexed FWALG(IV), FWALG(IV+1), FWALG(IV+2).

When FWALG is first calculated, it is calculated in the x-direction and the other directional components are initialised to this same value and modified later. The statements for this are:

$$\text{FWALG}(\text{IV}+1) = \text{FWALG}(\text{IV}) \qquad (4.5.1)$$

$$\text{FWALG}(\text{IV}+2) = \text{FWALG}(\text{IV}) \qquad (4.5.2)$$

The huge loop in which these and other such statements reside has a unit stride on loop index M. However, IV = VCTRLS(M) where array VCTRLS is a subset of numbers of the form K + 279*J, J=1,2,…,N. In RELAP5-3D, there is nothing remotely recursive about Statements 4.5.1 and 4.5.2. However, the compiler will not vectorize the loop that contains such statements. The compiler directives IVDEP and CONCURRENT do not overcome this.

The solution to this is to move transfer statements such as 4.5.1 and 4.5.2 to out of the loop. They were placed in a loop immediately after the huge loop. This removed the false recurrence problem.

4.6 Backward GOTO Inhibitor

In both PHANTV and PHANTJ, there is a backwards go to (go to 1521) that spans over 600 lines of code, 660 in PHANTJ and 1120 in PHANTV. Statement 1521 has a three clause if test whose body includes the backward go to and terminates on the next statement. The backward go to itself has a four clause if test for going backwards. In both subroutines, the backward go to can be executed no more than once per iteration of the huge loop.

The purpose of the backward jump is to provide smooth calculation of fiction and heat transfer coefficients for a volume/junction whose flow angle is in the transition region between vertical and horizontal. The first pass calculates horizontal or vertical coefficients. This is possible since the calculations are mostly identical for both cases with the main difference being the angle. The second pass is only

necessary if the flow angle is in the transition region. In this case, the second pass calculates the other type of coefficient to allow a smooth interpolation between them.

The compiler refused to vectorize the loop with the backward go to. This problem turned out to be much more difficult to solve than the problems with recurrence.

Duplicating 600+ lines of code would have created an unacceptable maintenance issue.

Replacing the backwards go to with a do-while loop failed because it was an inner loop of variable length.

Replacing the go to with a do loop inside the body of the 1521 if test and using the four clause if test to exit also failed. The compiler's messages were very obscure.

The solution was to create 2 subroutines, STRATV for PHANTV and STRATJ for PHANTJ, that comprised the entire bodies of the respective if statements at 1521. The subroutine would be called once and called a second time if the backward go to condition was true. This solved the maintenance problem, performed the interpolation, removed the backward go to, and eliminated the compiler objections.

It was not as simple as it sounds. There were over one hundred quantities calculated inside the new subroutines that had to be returned while numerous quantities were required as input. A common block was constructed to transfer most of them. It was found that if certain variables were included in the common block, the answers would be different for some input models, even though those variables had identical values on entry and exit whether they were calculated in STRATV/STRATJ or in PHANTV/PHANTJ. These were time consuming to find.

4.7 Branching Inhibitors

The worst problem for getting these huge loops to vectorize was the problem of deeply nested branches. The compiler messages were the most obscure when explaining why the huge loop could not vectorize when branching was the cause (or even that branching was the cause).

Branching affects vector performance and the deeper the nesting, the greater the effect. PHANTV has nine levels of nested if tests within its two huge loops; PHANTJ has 10. That does not count additional levels of branching introduced when subroutines, themselves having multiple levels of nesting, are inlined. For example, in PHANTJ, FIDISJ is called from a place 9 levels of nesting deep. FIDISJ has 6 levels of nesting and also calls subroutines with up to 3 levels of nesting.

The compiler could not inline all of these and still vectorize. The messages were hard to understand. At one point, all subroutine calls inside FIDISJ were commented out and turned on one by one until the compiler could not produce vector code. Then experiments with different subsets of the subroutine calls being activate were made. From this came another "discovery" about the effect of if tests on producing vector code.

The number of branches in an if test affects the compiler's ability to vectorize just as much as the level of deep nesting.

It seems that branches, beyond the first two, act as one level of nesting each. The reason for this may relate to the number of vector registers that the compiler has to work with and the way it assigns them among the conditional code.

Once this was understood, the strategy for overcoming deep nesting was immediately apparent: Reduce the number of branches and levels of nesting.

The means to do this is to create logical variables that combine the logical tests in if statements and else if clauses. In so doing, branches can be separated into independent if statements. Nested if tests can be moved out one or more levels.

For example, consider the following pseudo-code:

```
if ( void>0 ) then
  if ( void<1 ) then
    if ( btest(c,n) ) then
      CALCULATIONS 1
      if (hmap) then
        CALCULATIONS 1.1
      endif
    else
      CALCULATIONS 2
    endif
    ...
```

This can be converted to a single level of if test through the use of logical variables as follows:

```
LV0 = void>0
LV1 = LV0 .and. void<1
LVCN = LV1 .and. btest(c,n)
LVCNH = LVCN .and. hmap

if (LVCN) then
  CALCULATIONS 1
end if
if (LVCNH) then
  CALCULATIONS 1.1
end if
if (LV1 .and. .not.LVCN) then
  CALCULATIONS 2
end if
```

Both versions of the code produce the same calculations. The former is somewhat more readable, although using better names for the logical variables can actually make the second quite readable.

This example illustrates how four levels of nested if tests can be turned into one level with no branches (else if constructs). In PHANTJ and PHANTV, up to 6 levels of if tests were eliminated in this way. This technique was successively applied to nested if tests and if test branches until the compiler was able to vectorize the entire loop.

5. Speed-up Results

The vectorized results were expected to differ from the non-vector results because of the change in the calculation of the Burington angle. These changes in answers were within acceptable tolerances. The rest of the changes were

tested independently and verified to not change the results. Therefore, the overall code modification was deemed to be acceptable.

Timings were made of the code before and after the vectorization changes. In Table 5.1, the unmodified code is identified as Orig. and the modified and vectorized code is referred to as vector. The column marked V/O shows the ratio of vector MFLOPS and original MFLOPS. It can be seen that PHANTJ showed a speedup of 14 times its original speed for the 3DFLOW15 problem!

	PHANTV			*PHANTJ*		
Test Case	**Orig**	**Vect**	**V/O**	**Orig**	**Vect**	**V/O**
TYPPWR	13.5	57	4.2	10.9	66.5	6.1
ROSA	18.8	76.5	4.1	10.6	88.6	8.4
AP600	14.9	92.5	6.2	10.3	127.6	12.4
3Dflow15	16.0	98.5	6.2	9.7	136.1	14.0

Table 5.1 Improvement in subroutine MFLOPS

In Table 5.2, the time spent in the subroutines per call is listed. The column marked O/V shows the ratio of original call time to vector call time. The vector version of the code reduces the time spent in PHANTJ by up to a factor of 8.7.

	PHANTV Sec/Call			*PHANTJ Sec/Call*		
Test Case	**Orig.**	**Vect.**	**O/V**	**Orig.**	**Vect.**	**O/V**
TYPPWR	.00268	.00123	2.18	.00305	.00098	3.11
ROSA	.0132	.00333	3.96	.0149	.00398	3.74
AP600	.0654	.0156	4.19	.139	.0231	6.01
3Dflow15	.0482	.0119	4.05	.103	.0118	8.7

Table 5.2 Time spent in these subroutines per call

Generally, the speedup of PHANTJ is greater than that of PHANTV. One reason is that there is time spent holding issue. Another is the performance of numerous auxiliary loops in PHANTV. In PHANTV, there are about 30 small loops, some of which do not vectorize, while PHANTJ has none. All the speedup in PHANTJ is from the vectorization of one huge loop and all the work is in that loop. In PHANTV, some work is in loops that do not vectorize; this brings down the overall performance in accordance with Amdahl's Law.

The third and probably most important reason that PHANTJ runs faster than PHANTV when vectorized has to do with the difference in what each works on. PHANTV loops over the control volumes of the system and PHANTJ loops over the junctions. Junctions represent connections between control volumes and there are usually significantly more junctions than volumes. For multi-dimensional models such as 3DFLOW15, the number of junctions is more than twice that of volumes. This allows much longer vector length in the vector loops and results in higher MFLOPS.

The ultimate goal was to increase RELAP5-3D overall run speed by a factor of 50% by vectorizing these two subroutines and turning off the call to an optional subroutine, FORCES, that is seldom used and is not used in any of our test problems. The speedup for the whole code was as follows: 1.17 for 3DFLOW15, 1.22 for AP600, 1.43 for ROSA, and 1.51 for TYPPWR.

The speedup was greater for the smaller problems because PHANTV and PHANTJ took a larger percentage of the computational time in those cases. In the larger problems, more time is spent in the linear equation solver.

The goal of 50% speedup was actually exceeded for TYPPWR. This is mystifying at first since the percentage of time used by the three subroutines in question was only 33.2%. Even if the time taken by all three were reduced to zero, the speedup would still be less than 50%. The reason this happens is inlining of subroutines. Many subroutines with extremely low MFLOPS rates become part of a vectorizing loop in PHANTJ or PHANTV so that their operations also vectorize. In fact, these subroutines are not listed in the Perftrace timings after they have been inlined. The percentage of their execution time is subsumed into the execution time of the subroutine in which they were inlined. This is what caused the speedup to exceed 50%.

Conclusions

There are several conclusions to draw from this work. Legacy codes that were not vectorized in the early days of vector supercomputing should be re-examined. New developments in compiler technology, that have occurred after the early 1990's such as vectorization of a loop that contains a loop, may allow formerly non-vector loops to vectorize. Very long loops can be vectorized. Some of the techniques presented in this paper can be useful in vectorizing even extreme cases as were demonstrated here.

Acknowledgments

The authors would like to thank Bill Long of Cray, Inc. for his useful advice about modules and his suggestion to use MVBITS to further increase speed of certain operations. We would also like to thank the INEEL Chief Information Office for funding this work.

About the Authors

George Mesina has a Ph.D. in Mathematics and works in the thermal fluids department as a Senior Advisory Engineer. George does algorithm and code development for RELAP5-3D and RGUI and is responsible for RELAP5-3D code architecture and products. He can be reached at INEEL P.O. Box 1625, MS 3890, Idaho Falls, ID 83415 USA. Phone: 208-526-8612, email: mesinagl@inel.gov.

Peter Cebull is an Advisory Engineer on the HPC/visualization team at the INEEL. He has a background in nuclear engineering code maintenance and development and now serves as a Cray subject matter expert

in support of INEEL staff. He can be reached at INEEL P.O. Box 1625, MS 3605, Idaho Falls, ID 83415 USA. Phone: 208-526-1909, email: cebupp@inel.gov.