Evaluation of a Methodology for the Reverse Engineering and Parallelization of Sequential Code

Per H. Andersen*, Joseph Pizzi*, Runlin Zhu*, Youling Cao*, Donald J. Bagert*, John K. Antonio*, Fred Lott+, and John C. Grieger+

*Department of Computer Science
Texas Tech University
Lubbock, Texas 79409-3104

+Phillips Petroleum Company
Bartlesville, OK 74006

Abstract

A general methodology based on software engineering principles is proposed for the parallelization of existing sequential code. The utility of the proposed methodology is evaluated through a case study involving a numerically intensive application in the domain of petrochemical exploration. The methodology does not assume the existence of detail design documentation for the sequential code. The methodology involves three basic phases: (1) reverse engineering; (2) parallel design; and (3) parallel implementation. The process iterates between phases two and three until the values of the performance metrics satisfy project requirements. In addition to the methodology itself, considerable detail related to the experiences and lessons learned in performing the case study are included.

1: Introduction and background

Much progress has been made in the past five to ten years in defining middle-ware standards for parallel and distributed platforms, e.g., MPI and PVM [1, 12, 13]. Standards such as these make it relatively easy for parallel software developers to port their application software from one type of parallel platform to another, e.g., from a heterogeneous cluster of workstations to a massively parallel processor such as the IBM SP [3, 10, 11]. However, very little work has been published on defining and standardizing a software engineering based methodology for parallel software development itself.

A distinction can be made between how to define a methodology for *parallelizing* existing sequential code and a methodology for *developing* parallel software (i.e., "from scratch"). Even in cases where a sequential version of the application software exists, it could be argued that in order to achieve absolute peak parallel performance for a given application, one should not make extensive use of existing sequential code in the parallel software development process. The basis for such an argument is that the original developer of the sequential code never intended that the code be used as a basis for parallelization. It is possible that the way in which portions of the sequential design were originally encoded may actually hinder others from later discovering opportunities for parallelization. There are in fact examples in which the most efficient sequential algorithm for a particular computational task cannot be readily parallelized, but less efficient sequential algorithms for the same task may be extremely well-suited for parallelization. Thus, one can argue that using existing sequential code as a basis for parallelization may artificially constrain (or at least obscure) the design options available to the parallel software design process.

In theory, the above argument could be taken to the limit and the claim could be made that sequential code should not be used as a basis for developing parallel code. Instead, parallel software developers should re-think and re-engineer the entire application with parallelization in mind. In practice, however, one must acknowledge that there is a plethora of existing large-scale sequential applications that would benefit from parallel execution. Due to the associated time, cost, and general need for extensive domain expertise, it would be far too expensive (and thus impractical) to consider re-designing all such applications "from scratch." Many industries can derive significant profit, in the near term, from fast parallel execution of important applications - even if the parallel implementation is not fully optimized for peak performance. Thus, in practice there is a balance that must be struck between the time it takes to develop stable parallel code and that code's parallel efficiency.

The work described in this paper is motivated by the real need that exists for a well-defined methodology for parallelizing existing numerically intensive sequential applications. One such application, known as "elastic seismic wave modeling," is used here to help define and evaluate a general methodology for the parallelization of existing sequential code. During the initial process of defining the underlying methodological concepts presented, we were unaware of any existing similar methodologies that had been proposed and published in the literature. Later, however, we discovered a similar effort, known as SEMPA: Software Engineering Methods for Parallel Scientific Applications [7], which is based on a very similar framework to the one presented here. The similarity between our proposed methodology and the one independently developed by the SEMPA project reinforces its validity, intuitiveness, and utility. Another related effort that shares some concepts presented was developed as a result of the PARSE (Parallel Software Engineering) project [4].

The rest of the paper is organized as follows. In Section 2, the basic phases of the proposed methodology are defined at a high level. In Section 3, a case study involving the application of the proposed methodology to the parallelization of the "elastic seismic wave modeling" code is described. Section 4 contains a post analysis of both the code and the proposed methodology. This section highlights some of the lessons learned, outlines the strengths and weaknesses of the proposed approach, and indicates how the approach could be enhanced for future use. Some concluding remarks are provided in Section 5.

2: The proposed parallelization methodology

As illustrated in Figure 1, the proposed methodology involves three basic phases: (1) reverse engineering; (2) parallel design; and (3) parallel implementation. The input to phase one is the sequential code, and the output of phase one is called the reverse engineered for parallelization (REP) design document. The REP design document includes many of the characteristics of standard design documents for sequential code, but also includes extra features that are useful for doing parallel design. The REP design document serves as input for parallel design, which is done in phase two. Features of the REP design document that were particularly useful for the case study are described in more detail in Subsection 3.2.

The initial output of phase two is a prototype parallel design, which is then implemented in phase three. The initial prototype implementation is then evaluated using performance and other metrics. The information from this evaluation is fed back to phase two where refinements to the prototype design are developed. The process iterates between phases two and three until the values of the performance metrics satisfy project requirements. More details regarding the parallel design and implementation phases relevant to the case study are described in Subsections 3.3, 3.4, and 4.2.4.

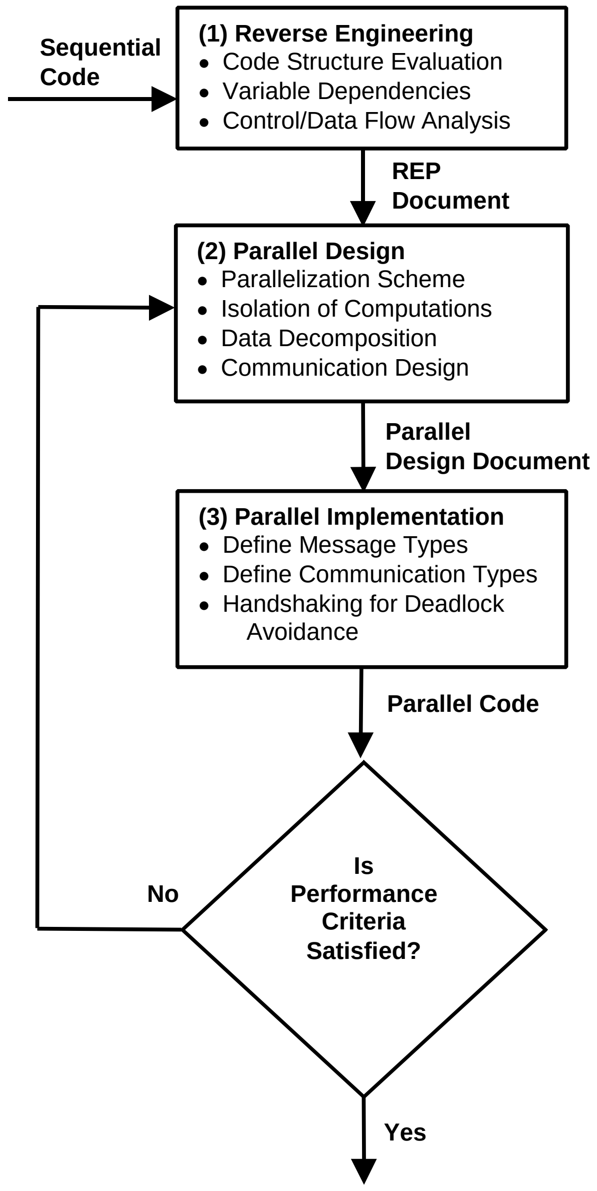

Fig. 1: Overview of the proposed methodology.

3: A case study using the proposed methodology

3.1: Overview of the elastic seismic wave modeling application code

Phillips Petroleum Company is actively exploring the subsalt oil in the Gulf of Mexico. With standard seismic processing, the seismic images under the salt are "blurred." The seismic image being blurred results in uncertainty as to where to drill. However, there exist seismic-processing algorithms (e.g., prestack depth migration) that greatly enhance the clarity of the seismic images under salt. These seismic algorithms, generically referred to here as elastic seismic wave modeling application codes, are very compute and data intensive.

A particular 2-dimensional sequential seismic wave modeling program, provided by Phillips Petroleum, was ported to the IBM SP machine in the Department of Computer Science at Texas Tech University. The team at Texas Tech signed a rigorous non-disclosure agreement prior to porting of this code. A main objective of this collaborative effort is to develop and evaluate relatively general software engineering techniques for serial to parallel program reengineering and redesign.

3.2: Reverse engineering

Reverse engineering can be described as the process of re-establishment of design. Its main tasks are analyzing the existing program, extracting design information from the program and forming a high level of abstraction, which will contribute the reengineering process. The REP design document has some characteristics that are quite different from ordinary design documents, but are very helpful for parallel design.

In order to obtain a practical strategy, the team must examine the source code, reconstruct or remodel it so that the program structure can be perceived, and derive an efficient format of design that benefits the process of parallel design.

There were three stages to the task of reverse engineering of the given sequential program. The first stage was the identification of the source code logic. This was accomplished by performing a procedural design for the original code, by first numbering the lines of the source code, then giving detailed descriptions for all services and finally drawing a flow diagram for the unstructured source code. Though the level of abstraction was somewhat low, this step constituted the basis for analysis and obtaining a higher level of abstraction of information for the program.

In the second step, in order to facilitate the reengineering of the software for a parallel system, the source code was analyzed and the abstraction simplified by creating a format that would facilitate the parallelization process. The relationship between the functions and variables were highlighted by using sufficiently abstracted pseudocode. In the variable description list, we traced and dissected data separately by listing their attributes, types, statuses, and critical points for the bound checking of functions. Because data dependency is also important for processing data in parallel systems, for every variable, all other variables that have a logical connection to it were listed. For the convenience of parallelizing, the points in the sequence of logic flow where data are referred were also listed, and flags within the program that show important information data flow were identified.

In the final stage, the abstractions from the final specification were refined and the data control flow diagram that had become the starting point of the reengineering procedure was constructed. Because global data played a dominant role in the given program, the diagram clearly described the global variables in term of updates and references. This diagram also adds to the knowledge of variable dependencies and assists in making decisions on how to setup data sharing or data splitting in the parallel design.

The documentation obtained in the reverse engineering phase is not finalized until the whole process of reengineering is complete. Any more specific information is added following the process of reengineering; this is especially true for the user interface and file structures. The result of the reverse engineering process is a REP design document which can be used in the subsequent process of reengineering [9, pp. 770-1].

3.3: Prototype parallel design

The parallel platform targeted for this study was the IBM SP SuperScalar system, which consists of multiple independent processing nodes interconnected by a high-speed interconnection network [11]. Because each node is basically a stand-alone machine (i.e., a workstation minus the keyboard and monitor), the IBM SP architecture is described in [8, p. 29] as a “cluster-in-a-box.” However, the speed and scalability of the interconnection network of the IBM SP distinguishes it from a typical LAN-based cluster of workstations.

The IBM SP (as well as clusters of workstations) could be used to implement task parallelism, in which functionally independent tasks are executed on distinct nodes in parallel. However, for numerically intensive applications like the one studied here, it is typically more fruitful (and easier) to exploit data parallelism. The concept of data parallelism is to divide the application’s data set across the nodes so that each node performs calculations using its local data partition. This style of parallelism, which is implemented by having all nodes execute the same program on distinct data sets, is known as the single program, multiple data (SPMD) style of parallel programming. In using the SPMD approach, it is usually the case that intermediate results must be exchanged among the nodes as the parallel computation progresses. The way in which the data set is partitioned can impact the volume and timing of the inter-node communications that are required. Because excessive inter-node communications can degrade overall performance, understanding the data dependencies for an application is a key element in deciding how to best partition the data across the nodes.

The first step in the prototype parallel design process was to determine the portions of the sequential code in which the computational load was concentrated. The sequential program was instrumented and timings were taken on all areas in which intensive computations appeared to be likely (e.g., around nested looping constructs). The overall execution time was also monitored in order determine if any significant work was overlooked.

Once the work-load concentrations were determined, it was possible to go back to the data dependency diagrams (defined in REP design document) and determine which significant variables were involved. The dependencies among these and other variables were then analyzed. A variable flow diagram was developed as a mechanism for identifying all the points in the model where data dependencies had to be accounted for in the parallel design. Again, the data dependency diagrams were useful in reducing the effort required in this stage of design. From the variable flow diagrams it was possible to map out the message passing requirements for the parallel design.

An important part of the structure of the sequential program was an "outer" iterative loop. For each outer loop iteration, several inner nested loops were executed in sequence (within the outer loop body). There were strong data dependencies among the variables updated within consecutively executed inner loops, so executing the inner loops in parallel was not possible. Instead, it was decided to parallelize each nested inner loop.

In general, each nested inner loop updated the values stored in a 2-dimensioanl array. For some inner loops there were data dependencies between columns only, i.e., updating the values in the *i*-th column required data values form the (*i* – 1)-st column. Other inner loops had data dependencies among the rows only, and still others had data dependencies among both rows and columns. Figure 2 shows a typical data decomposition scheme used for parallelizing an inner loop. This type of data decomposition diagram was very useful in describing the parallel design.

Because the parallelization of some inner loops required inter-node communications, a "space-time" diagram was used to document the design of the required communications. Figure 3 shows such a diagram. The term "space-time" relates the fact that the data is divided over space (i.e., among nodes of the distributed memory system) and the horizontal direction indicates the computations and communications that take place as time progresses. It is important to note that this diagram was *not* produced using postmortem data captured

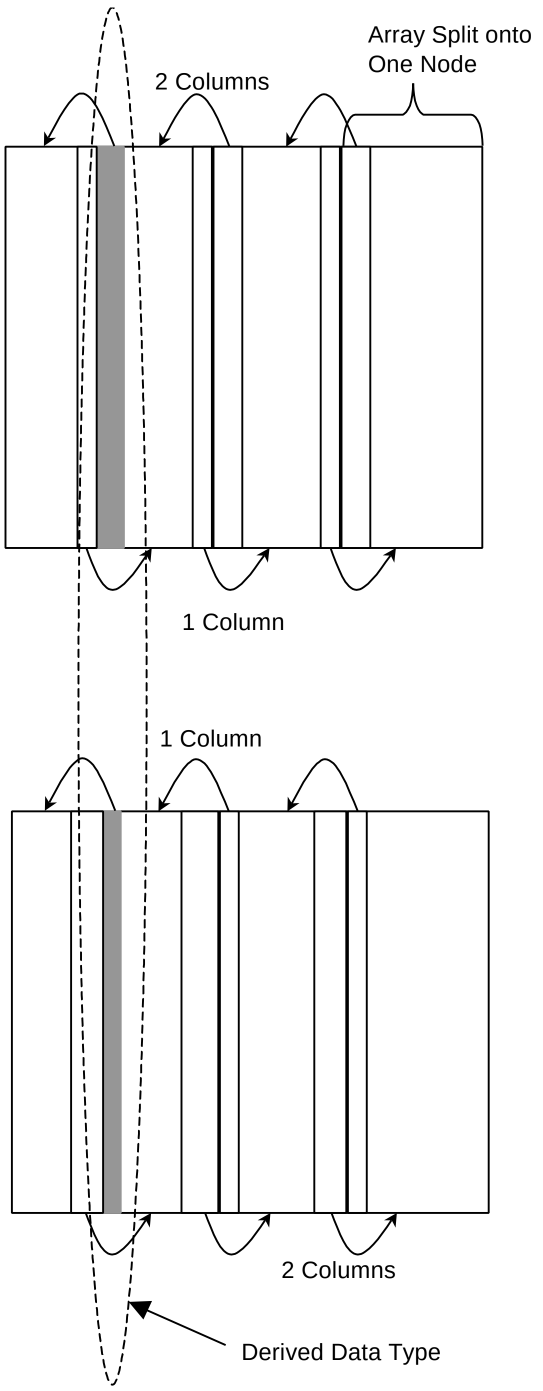

Fig. 2: Typical array splitting technique, showing examples of data movement and a derived data type. (Reported in the parallel design document.)

during parallel execution. The diagram in Figure 3 is a part of the *design* process; it illustrates the designer's estimate of the relative timings and relationships between computations and communications. There are, in fact,

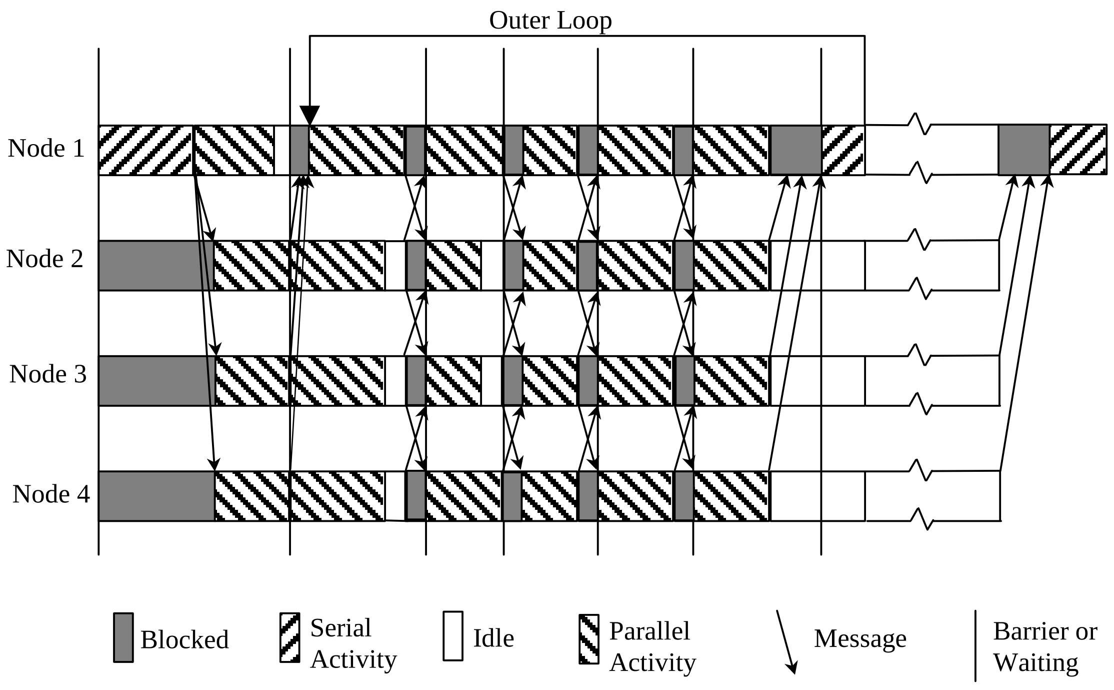

Fig. 3: Typical space-time diagram. (Reported in the parallel design document.)

postmortem tools such as VT [3] that can be used to analyze, precisely, timings associated with an actual parallel implementation and resulting execution. However, at this stage of the process, the code is not implemented, so such a tool was not applicable.

3.4: Prototype parallel implementation

Because the original serial program was written in FORTRAN, the data parallelism was implemented by sharing the data arrays column-wise. The points in the program where message passing was required was implemented with calls to non-blocking message passing procedures. The non-blocking calls were implemented since at these points in the program, messages had to be sent both to the previous and next node as well as messages received from both the previous and next node. Deadlock is a potential problem with this kind of message passing structure. Deadlock was avoided by using non-blocking calls coupled with a call to a message passing procedure, which waits for all messages to complete [1]. In addition to implementing the message passing, the array indexing on each node had to be adjusted to account for the reduced number of calculations required on each node.

The structures of the messages were, generally, a derived data type. MPI was used, which makes it possible to derive data types when data is not contiguous in memory. This mechanism reduces the number of times data is copied between memory locations [13]. A careful analysis of the data dependencies made it possible to determine which variables could be collected together and sent as one message even before a variable might be required by the program. This was done in an attempt to keep the number of messages to a minimum.

The careful attention to Software Engineering techniques and design documentation resulted in a parallel program that was implemented in one step [2]. Except for a few typographical errors, the parallel program ran within one hour of being completed and no design changes were required. The original sequential program was around 940 lines of code and the prototype parallel program was about 1,740 lines of code. Although further work related to optimization of both the serial and parallel programs will be investigated in the future, the prototype parallel implementation preformed better than expected. Performance and timing details are provided in the next subsection.

The prototype parallel program was tested on the IBM SP at Texas Tech University, consisting of 4 nodes. Eventually the code will be moved to the IBM SP system operated by Maui High Performance Computing Center (MPHCC) and the number of nodes increased to 64.

3.5: Refinements of parallel design and implementation

3.5.1: Establishing a means for evaluation. Several obstacles had to be overcome before the program used in this study could be further refined. The program is a 2-D seismic model, which occupies a position of importance within the petroleum industry. Not only is the algorithm guarded but so are the data sets used to initialize the model as well as the results generated by the model. Phillips Petroleum provided Texas Tech with one set of initial conditions, and this data set was fairly small. Because of the lack of a large set of initial conditions to test the model with, the initial data set was expanded (synthetically) in steps to create multiple data sets of larger sizes for testing and evaluating the performance of the parallel implementation.

A second obstacle that had to be overcome was the lack of an optimizing preprocessor on the Texas Tech IBM SP. Code refinement on an IBM SP involves optimizing the programs for the IBM SP POWER2 processor. Although the IBM SP XLF compiler optimizes code fairly well, the recommended [6] approach to optimizing code is to take advantage of a third party preprocessor such as the KAP preprocessor from Kuck and Associates [5]. The Texas Tech IBM SP does not have a preprocessor, so code refinement was implemented using the SP XLF compiler. An IBM SP system account request has been made to the Maui High Performance Computer Center in order to take advantage of the preprocessors on their systems for future studies.

A third obstacle was limited access to a domain specialist during the reverse engineering and parallel design phases. This “obstacle” actually served as a benefit for this study. One of the goals of the study was to determine whether a domain specific model like the one used could be reverse engineered and parallelized using software engineering techniques alone. Only twice during the parallelization would closer contact with a domain specialist have helped. The first instance was a point in the code where a serial dependency occurred at the boundaries within an array. A decision was made to use data from the previous iteration at the boundaries, resulting in a difference in the final result at the fifth decimal point in the mantissa. The other instance that might have been improved with input from a domain specialist was the fact that in the sequential program intermediate results are collected and dumped to disk during execution. An insight on how this data was being used might have resulted in a better parallel design. In this case it was decided that each node would return its intermediate results to a master node, which would dump the results to disk.

3.5.2: Performance results. The performance of the initial parallel implementation was very impressive with all combinations of data set sizes and number of nodes used. The original sequential code executed using the initial (small) data set resulted in an execution time of about 3.6 seconds on a single SP node. This was based on compiling the original sequential code without any of the compiler optimization switches set. The parallel program accomplished the same calculation on four SP nodes (on the same data set) with an execution time of about 0.9 seconds. Again, none of the optimization switches of the compiler were set.

A test on the largest data set resulted in a sequential execution time of 30.25 seconds, again with no optimization switches set. The large data set was obtained by decreasing the grain size of the simulation up to the memory limitations of a single SP node, which was 256 Mbytes.

The effect of setting various compiler optimization switches were then studied. Table 1 provides a list of the options and a brief description. For more information on the IBM SP XLF Compiler see Katherine Stewart’s article on optimizing with the XLF compiler [10].

Table 1: Descriptions of compiler optimization switches.

Switch Label	Description
O3	Memory and compiler time intensive optimizations
hot	Performs high order transformations on loops
arch	Make use of instructions specific to a given processor
tune	Tune the code for a specific processor
cache	Customize the code for a specific cache configuration

The following is an example of a compile command line that makes use of all of the switches listed in Table 1.

```
xlf -O3 -qhot -qarch=pwr2 -qtune=pwr2
–qcache = level = 1:type = d:size =
128:line = 256:assoc=4:cost=12 –qcache
= level = :type = i:size = 32:line =
128:assoc = 2:cost = 12 -o program
program.f
```

Table 2: Serial execution time on one SP node.

Compiler Switches Used	Execution Time (sec.)
none	30.25
O3	14.25
O3 and hot	11.40
O3, hot, and arch	11.23
O3, hot, arch, and tune	11.18
O3, hot, arch, tune, and cache	11.06

Table 2 shows the execution time results of applying various combinations of the compiler switches for the sequential code. Table 3 shows the execution time results of applying various combinations of the compiler switches for the parallel code executing on four SP nodes. The data set used was the same as that used for collecting the sequential execution times of Table 2.

Table 3: Parallel execution time on four SP nodes.

Compiler Switches Used	Execution Time (sec.)
none	6.98
O3	2.75
O3 and hot	2.74
O3, hot, and arch	2.72
O3, hot, arch, and tune	2.68
O3, hot, arch, tune, and cache	2.67

From the execution time results tabulated above, the speedup for the data set tested is more than four, which is the number of SP nodes used for parallel execution. The reason for this "super-linear" speedup is related to the fact that the large size of the data set used combined with the nested looping structure of the application resulted in poor cache utilization and thus large numbers of memory references for sequential execution.

One of the strengths of the preprocessor tools is their ability to effectively optimize code for nested loops by improving cache utilization and reducing memory references. The type of large and deeply nested loops found in this particular simulation model are not easily optimized by the IBM SP XLF compiler. Experiences on the IBM SP with matrix multiplication algorithms have shown that hand coded loop unrolling and cache blocking for complex loop structures is more productive in obtaining performance gains than what can be expected from the XLF compiler. Again, as stated in the previous subsection, further studies will be conducted when access to a third party preprocessor is obtained.

4: Post analysis

4.1: Code

A post development evaluation of the code was done. Two code evaluation tools were used, Verilog's Logiscope [15] and Paralogic's Bert 77 [14]. Both tools were demo evaluation copies although Texas Tech University has since purchased the license for Logiscope.

4.1.1: Logiscope. Verilog describes its Logiscope tool as a test and maintenance tool for C and C++ programs. Some of the key features of Logiscope are: a call graph used for illustrating the overall architecture of an application by displaying the calling relationships in an application; a control graph that illustrates the logical structure of a component of an application (e.g., a function); and a quality model similar to ones defined by Boehm and McCall [15]. The actual implementation of the software quality analysis is via a series of Kiviat graphs, a test mechanism, and a standard set of metrics for measuring the complexity of code is also included in Logiscope.

Before Logiscope could be used, the Fortran program had to be converted to C or C++. There were two approaches available for translating the program: by hand coding the translation or by using a Fortran to C converter. The Fortran to C converter approach was chosen to save time. F2c, an AT&T, Lucent Technologies and Bellcore product was chosen as the converter. It was installed on the IBM SP and the model was then converted to C. The Logiscope product is a Windows application therefore the analysis of the converted program was run on an NT workstation. The program was run through Logiscope primarily as a method of exposing any gaps in the reverse engineering and parallel design of the model using software engineering techniques. A secondary reason was part of the evaluation of Logiscope as a tool for software analysis. We wanted to determine if the methodology, of converting Fortran programs to C, for Logiscope analysis is a valid methodology.

The result of the Logiscope analysis was the confirmation that nothing significant was overlooked during the reverse engineering and parallel design. An example of a call graph generated by Logiscope for the model is illustrated in Figure 4. A window containing the source code combined with the call graph is required to understand the significance of the call graph. From the figure, a large loop is apparent with a number of smaller nested loops that have calls buried in them. Some of the calls represent functions with additional nested loops in them. A complete picture of the loops and the call sequences can be obtained from the Logiscope tool.

The methodology of converting Fortran to C and applying Logiscope to the C code has potential but the results are inconclusive. The call graphs and control graphs were generated without too much trouble with this approach. However, the structure of the code was fairly easy to determine using manual analysis techniques for this study. For larger applications, utilizing an

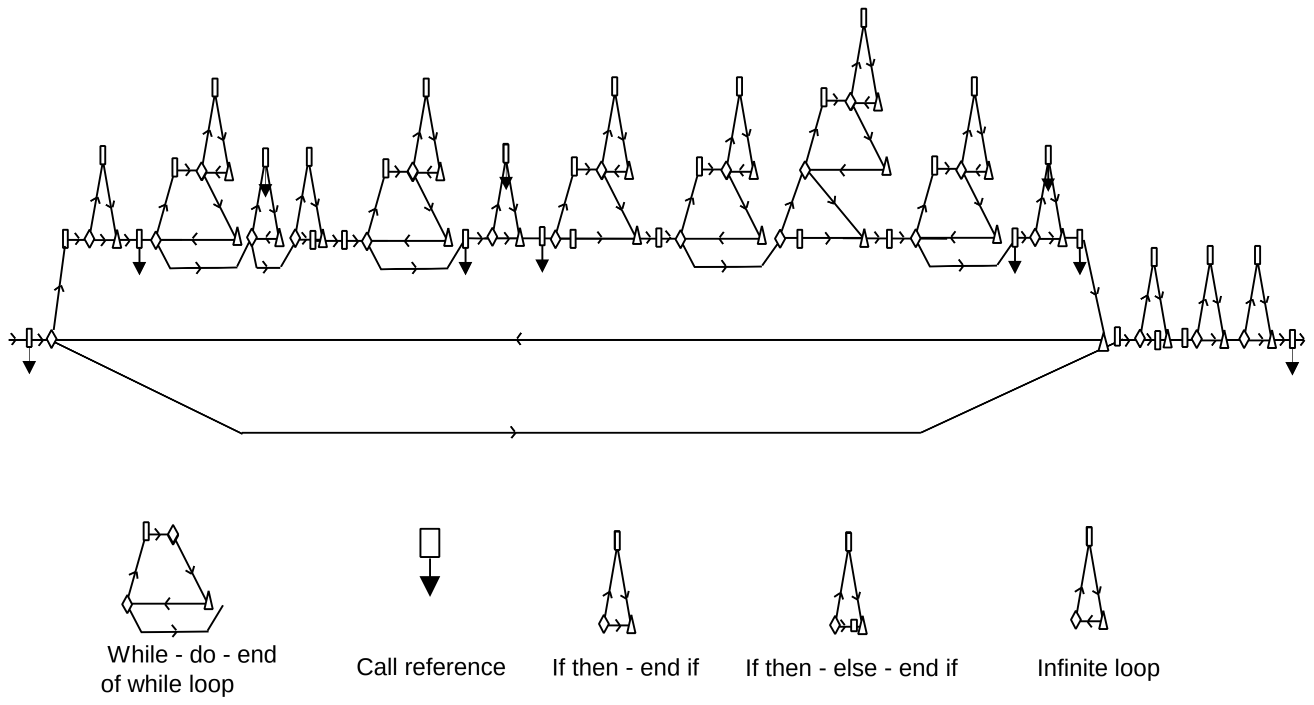

Fig. 4: Sample call graph generated by Logiscope for main function.

automated approach (i.e., using a tool like Logiscope) may be more important or worthwhile.

4.1.2: Bert 77. The functionality of Bert 77 is quite different from that of Logiscope. Paralogic describes Bert 77 as “an automatic and optimizing parallelizer for FORTRAN 77.” From [14], Paralogic claims that Bert 77 aids in answering the following questions:

- *Is my application concurrent?*
- *How much performance increase can I get from running my application on a cluster of workstations? a shared memory machine?*
- *If I make my application concurrent, how much faster will it really go?*
- *What is the effect of adding more processors?*

There are two ways in which Bert 77 can be run: the recommended method is via an X-windows GUI; the other method is via command line. Both methods were used for the analysis. The goal of Bert 77 is to expose any potential areas within the program for parallelization that might be overlooked. Unlike Logiscope, Bert 77 makes recommendations on what can and can’t be parallelized and the efficiency of parallelizing. The amount of data generated by programs like Logiscope and Bert 77 can be daunting. For example, the sequential program used in this study is moderate in size (less than 1000 lines of code). However, Bert 77 generated 69 pages of data during its analysis of this code.

Figure 5 is a small excerpt from the output of Bert 77 when used in the command line mode. From the figure, both Bert 77 and the parallel program design were in agreement with respect to the first loop being a poor choice for parallelization and the second loop being a potential candidate for parallelization.

Bert 77 lists loops that are potential candidates for parallelization as concurrent and lists loops that are poor candidates for parallelized as not concurrent. The only serious problem with the results from the Bert 77 analysis was the poor efficiency values it generated. This is probably related to the fact that a version of Bert 77 is not yet available for the IBM SP, therefore in order to do the analysis, a Linux version was installed on a standalone PC. If the efficiency values are ignored, Bert 77 actually did a fair job. It detected 75 loops and determined 61 loops were concurrent and 14 loops were not concurrent. Of the 61 concurrent loops 19 were in sections of code that had very low workload concentrations, the same was true for 10 of the 14 loops that were not concurrent. That leaves 42 concurrent loops and 4 not concurrent. In the actual parallel program developed for this study, 24 of the loops detected by Bert 77 as concurrent were parallelized and the four not concurrent loops were not parallelized. The remaining 18 loops detected by Bert 77 as concurrent tended to be inner loops of nested loops which were not parallelized due to the data partitioning scheme selected for the parallel program design.

```
------------------------------------------
/home/andersen/phillips/elastic3.f:
LINE: 125
- - - - - - - - - - - - - - - - - - - - -
   ACTION:
          Processing DO loop.
   NOTE:  Loop contains file operations
   RESULT:
          Loop is not concurrent
------------------------------------------

/home/andersen/phillips/elastic3.f:
LINE: 133
- - - - - - - - - - - - - - - - - - - - -
   ACTION:
          Processing DO loop.
   RESULT:
          Loop is concurrent

- - - - - - - - - - - - - - - - - - - - -
   DO LOOP BODY
      IN = {nx, dtx, a1, a2, pz1(3: nx-3,
      k_BERT_LOW-1: k_BERT_UP+2),
      xy1(3-1: nx-3+2, k), v1(3: nx-3, k),
      dens(3: nx-3, k)}
      OUT = {j, v1(3: nx-3, k)}
- - - - - - - - - - - - - - - - - - - - -
   ESTIMATIONS:
      Execution time: 0.013552 seconds
        repeated 14900.0 times  (17.6%)
      Flow overhead time:0.021052(seconds)
      Dynamic overhead time: 0.020964
        (seconds)
      Estimated number of iterations : 298
       NOTE: Efficiency of DO loop depends
         on boundary values
      Dataflow speedup: 0.58367 times with
         8 processors(-12.5563%)
      Dataflow parallelization is
         estimated as: INEFFICIENT
      Dynamic speedup:0.58589 times with 8
         processors (-12.4418%)
      Dynamic parallelization is estimated
         as: INEFFICIENT

------------------------------------------
```

Fig. 5: Example excerpt from Bert 77 output analysis file.

4.1.3: Summary of Logiscope and Bert 77. Both Logiscope and Bert 77 were useful tools for analyzing the sequential program. Both packages tend to produce a lot of information leaving it up to the programmer to decide on how to make use of the results. Logiscope is not a Fortran programming tool so probably should be used with caution. Bert 77 appears to have potential. Its inability to provide good efficiency values will be investigated once an IBM SP version becomes available.

4.2: Methodology

A post-implementation analysis of the software methodology used to develop the parallelized software from the existing sequential code in the case study provided some useful information (described below) that will be helpful for future work in this area.

4.2.1: Team software development and project planning. Previously, parallelization of sequential code often involved a single individual. In this case study, a team of four graduate students, supervised by two faculty, reverse and re-engineered sequential code. There were two graduate students and one faculty member whose primary research area is software engineering, and two graduate students and one faculty researcher in high performance computing as well. The goal was to have a working prototype within 22 calendar days after starting the project. The usual response to such a short deadline is to forego the proper infrastructure to have an effective software process; however, studies have shown that even in such cases, good project planning and effective software quality assurance can significantly reduce the total development time [2].

4.2.2: Team software development and inspections. A team of four developers is ideal for design document and code inspections. Not only did the inspections significantly reduce testing time, but also allowed for greater communication and sharing of ideas among the team members.

4.2.3: Individual software development. A software team is most effective when each of its individuals is a strong software developer. The Personal Software Process^{SM} (PSP^{SM}) strategy [2] is a good starting point for developers to each define an individual software development process while improving their skills. At the time of this case study, only one of the four developers had been trained in PSP^{SM}; the other three are currently going through a course using the process. All of the developers agree that using some of the PSP^{SM} techniques for gathering data and in the design and implementation of the parallel code were useful.

^{SM} Personal Software Process is a service mark of Carnegie Mellon University.
^{SM} PSP is a service mark of Carnegie Mellon University.

4.2.4: The parallel design document. As the initial version of the parallel design document was being created, it was evident that some specific formalisms were needed. The parallel design document consisted of: a space-time diagram (Figure 3), a message passing tracking table (associates variables with source code line numbers where message passing of listed variables must occur), message structure diagrams (Figure 2), a control flow diagram for primary variables, and the dependency forms and pseudo-code from the reverse engineering document. These design document features proved to be useful to varying degrees. The signal flow diagrams, similar to the Logiscope control flow diagrams, were the least useful; the dependency forms provided the same information in a more useful format that was easier to generate. The most useful tool in the parallel design document were the message passing tracking tables and message passing structure diagrams. They were used in conjunction with the pseudo-code as a guideline during the actual coding of the parallel program. The space-time diagram was useful during program debugging, but also helped in solidifying the parallel design concept at a fairly high level of abstraction.

From our experience, the parallel design document might ideally include: a space-time diagram, a message passing tracking table, message structure diagrams, the variable dependency forms, pseudo-code, control flow diagrams from Logiscope and loop concurrency analysis from Bert 77.

5: Conclusions

Based on the results of the case study, the proposed methodology is deemed to be a solid foundation for the parallelization of sequential code. The process of going from sequential code to the prototype parallel implementation required approximately 120 person-hours, and was accomplished over a period of about a month. About 80 hours were spent on reverse engineering, 20 hours on parallel design, and 20 hours on parallel implementation. The 80 hours counted as "reverse engineering" time includes about 40 hours that was devoted to determining exactly what the REP document needed, and what it did not need, i.e., defining the details of the methodology itself. Thus, discounting the time required for defining the details of the reverse engineering process and documentation, only a total of about 80 hours were spent on the actual process of parallelizing the code. The performance of the parallel implementation was very good. Future work includes tuning the methodology based on the insights reported in Section 4 (Post analysis) of this paper and attempting to apply the methodology for parallelizing another sequential application.

References

[1] Gropp, W., Lusk, R., and Thakur, R, "Introduction to Performance Issues in Using MPI for Communications and I/O," *Tutorial from the7th IEEE Symposium on High-Performance Distributed Computing*, Chicago, July 1998.

[2] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, Reading, MA, 1995.

[3] *IBM Parallel Environment for AIX: Hitchhiker's Guide*, Version 2, Release 3, order no. GC23-3895, International Business Machines Corporation, Poughkeepsie, NY, 1997.

[4] Jelly, I., Gorton, I., "The PARSE Project," *Software Engineering for Parallel and Distributed Systems*, edited by I. Jelly, I. Gorton, and P. Croll, Chapman & Hall, New York, NY, 1996.

[5] "KAP for IBM Fortran and C," Kuck & Associates, Inc. Champaign, IL, http://www.kai.com.

[6] Lantz, S. "Single-Processor Performance Considerations for the SP2," Cornell Theory Center, http://www.tc.cornell.edu/Edu/Talks/Performance/SingleProcPerf/, 1998.

[7] Luksch, P., Maier, U., Rathmayer, S., Weidmann, M., "SEMPA: Software Engineering Methods for Parallel Scientific Applications," *Software Engineering for Parallel and Distributed Systems*, edited by I. Jelly, I. Gorton, and P. Croll, Chapman & Hall, New York, NY, 1996.

[8] Pfister, G., *In Search of Clusters*, Second Edition, Prentice Hall, Upper Saddle River, NJ, 1998.

[9] Pressman, R., *Software Engineering: A Practitioner's Approach*, 4th Ed., McGraw-Hill, New York, NY, 1997.

[10] Stewart, K., "Using the XL Compiler Options to Improve Application Performance," http://www.rs6000.ibm.com/resource/technology/options.html, 1999.

[11] Stunkel, C., *et al*, "The SP2 High-Performance Switch," IBM Systems Journal, Vol. 34, No. 2, 1995.

[12] Wilkinson, B. and Allen, M., *Parallel Programming*, Prentice Hall, Upper Saddle River, NJ, 1999.

[13] Pacheco, P., *Parallel Programming with MPI*, Morgan Kaufmann, San Fransico, CA, 1997.

[14] "BERT 77: Automatic and Efficient Parallelizer for FORTRAN," http://www.plogic.com/bert.html, 1998.

[15] "Logiscope C/C++ WinViewer," VERILOG, Inc., Dallas, TX, document reference: D/LEWX/MA/000/740, http://www.verilogusa.com, 1997.