FCRASH, Optimization of an Explicit FEA Based Crash Simulation Code For The Cray C916/16-512

Daniel F. Anderson and *Alexander Akkerman*, Ford Motor Company, Dearborn, MI and *Dave Strenski*, Cray Research Inc., Dearborn, MI

ABSTRACT*: Starting out as a small research project within Ford Motor Company in 1987,* FCRASH *has grown into a fast and viable transient dynamic analysis finite element program for vehicle crash simulation. Most commercially available finite element crash analysis packages evolved from an impact simulation code, DYNA-3D, originally written at Lawrence Livermore National Laboratories. FCRASH, however, was written from scratch to provide a more structured program capable of greater computational efficiency and ease of maintenance. Part of FCRASH's success was the result of a close collaboration between Ford Motor Company engineers and consultants from Cray Research. Our paper will focus on this collaborative effort, and will discuss the combination and trade-offs of vector and parallel optimization techniques that enable FCRASH to be the fastest vehicle structures crash simulation program within Ford.*

INTRODUCTION

FCRASH is an explicit finite element (FEA) analysis code that consists of about a quarter of a million lines of code, roughly 90% FORTRAN and 10% C, distributed among 1,400 subroutines and functions. FCRASH is one of the FEA solvers used in conjunction with a set of tools at Ford Motor Company that aid in the design of vehicles and components for crash worthiness. The code is now used in production within several divisions of Ford, and is currently running on seven different platforms, with the majority of the large vehicle models being run on the CRAY-C916/16-512 systems located in the Engineering Computing Center (ECC).

Most recent studies in optimizing large scale scientific and engineering programs have focused on emerging massively parallel processing (MPP) technologies. However, traditional shared memory vector supercomputers such as the CRAY-C90 still perform the majority of Computer Aided Engineering (CAE) simulations at automotive engineering and manufacturing companies such as Ford. It comes as no surprise that today's supercomputer user community expects highly efficient single processor performance from a vector supercomputer. At Ford we find our user community also demanding parallel implementations of their simulation runs even though they can only expect modest throughput improvements on a busy 16 processor machine.

It has been common knowledge for some time that finite element simulation codes are rich in potential parallel constructs[1]. Parallelism within a simulation code exists at several levels, from the job or program level to the instruction level[2]. For a single simulation run on a parallel vector computer parallelization across the task or procedure level and parallelization at the instruction level (i.e. vectorization) provide the most important paths to performance improvement.

Before describing the work done to optimize the FCRASH code it is useful to review the fundamentals of an explicit time integration based finite element program.The purpose of the review is to analyze the structure of the program on the CRAY-C90, with the focus on optimization, many of the necessary details are omitted.

Velocity and Acceleration central difference formulas

$$\dot{u}^{n+1/2} = \frac{u^{n+1} - u^{n}}{\Delta t}$$

$$\ddot{u}^{n} = \frac{\dot{u}^{n+1/2} - \dot{u}^{n-1/2}}{\Delta t}$$

Equation of motion

$$M\ddot{u} = f = f_{ext} - f_{int}$$

Velocity and Position update formulas

$$\dot{u}^{n+1/2} = \dot{u}^{n-1/2} + \Delta t^{n} M^{-1}(f_{ext}^{n} - f_{int}^{n})$$

$$u^{n+1} = u^{n} + \Delta t^{n+1/2}\dot{u}^{n+1/2}$$

Time Step

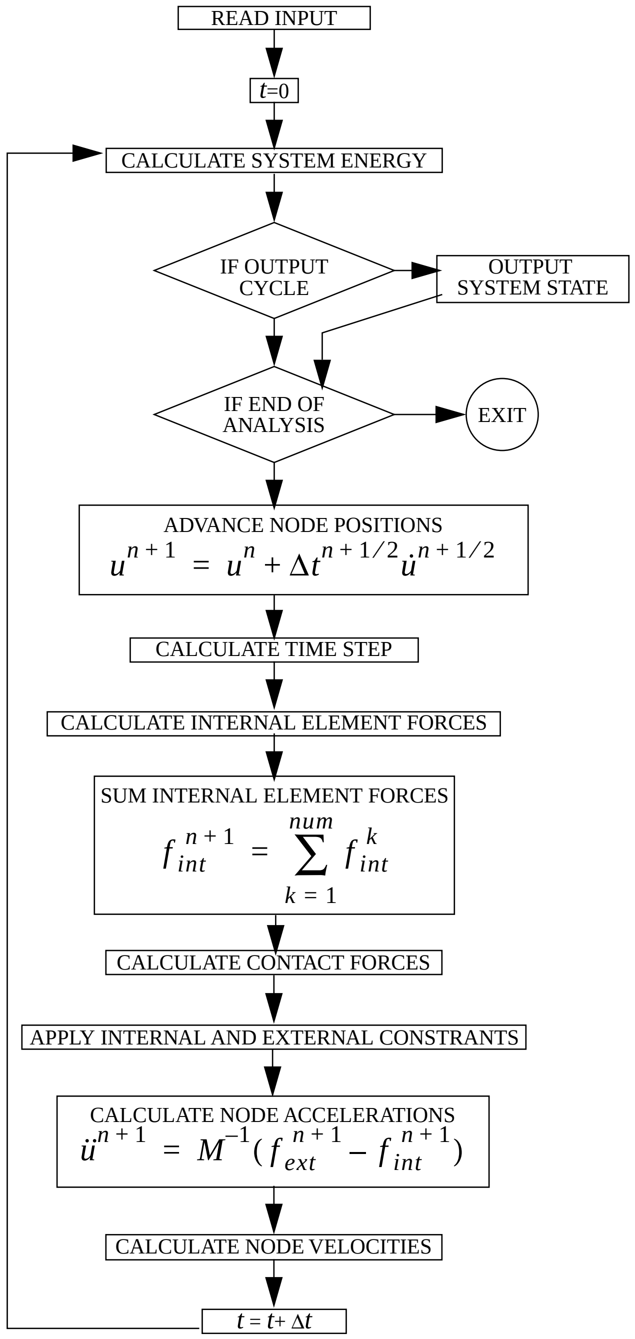
$$\Delta t^{n} = (\Delta t^{n-1/2} + \Delta t^{n+1/2})/2$$

Internal Element Force

$$f_{int} = \int_{V_c} B^{T}\sigma dV$$

Where: u = nodal displacement
f_{int}, f_{ext} = internal and external forces
σ = element stress
B = shear displacement matrix
M = mass matrix

The main integration loop is summarized in the flowchart outlined below. Calculation of the element forces provides the



READ INPUT
t=0
CALCULATE SYSTEM ENERGY
IF OUTPUT CYCLE
OUTPUT SYSTEM STATE
IF END OF ANALYSIS
EXIT
ADVANCE NODE POSITIONS
$u^{n+1} = u^n + \Delta t^{n+1/2} \dot{u}^{n+1/2}$
CALCULATE TIME STEP
CALCULATE INTERNAL ELEMENT FORCES
SUM INTERNAL ELEMENT FORCES
$f_{int}^{n+1} = \sum_{k=1}^{num} f_{int}^{k}$
CALCULATE CONTACT FORCES
APPLY INTERNAL AND EXTERNAL CONSTRANTS
CALCULATE NODE ACCELERATIONS
$\ddot{u}^{n+1} = M^{-1}(f_{ext}^{n+1} - f_{int}^{n+1})$
CALCULATE NODE VELOCITIES
$t = t + \Delta t$

greatest opportunity for both vectorization and parallelization in the explicit FEA formulation. Fortunately in a typical crash simulation the force calculations represent 60% of the computational effort. The element calculations are entirely independent and thus inherently parallel. Element force calculations performed on elements of the same type and material can be readily vectorized and the fact that the majority of elements in a structural crash simulation typically consist of elastic-plastic shells, makes vectorization a particularly useful optimization tool.

Because an explicit formulation does not require the formation of a global stiffness matrix, and the need for out of core solutions, I/O operations represent only a very small percentage of the over all simulation time. I/O is used only to store the state of the model at regular intervals for post-processing purposes. However, because of our goal for 30 minute job completion we were forced to optimize this I/O performance as well.

It should be noted that contact-impact operations must be performed in any crash simulation code, as shown in the flow chart by "Calculate Contact Forces". Although these operations are performed in parallel in the current version of FCRASH they will not be discussed in detail in this paper. We will concentrate on the vectorization and parallelization techniques used to speed up the force calculations as well as mention some I/O improvements which were applied.

We will trace performance improvements starting with the CRAY-YMP version in 1992 to the current CRAY-C916/16-512 first released at the end of 1994. As the code was being optimized, it is important to note that the development of new code features continued. This compounded the difficulty of the optimization task as these new features often required additional code restructuring to maintain computational efficiency. Finally, optimization of the code had to be structured in a manner which did not interfere with code portability. FCRASH is routinely released across seven computer platforms. During the early development stages full releases were produced every three months.

As shown in the following bar chart (see Figure 2), the performance of FCRASH initially running on the CRAY-YMP was 12.4 hours for a 100 milliseconds full frontal impact simulation of a midsize car modeled with about 22,000 elements (see Figure 1). By the end of this effort this same model was executed in under 20 minutes on a CRAY-C916/16-512, a total speedup of 40 times in over a two year period.

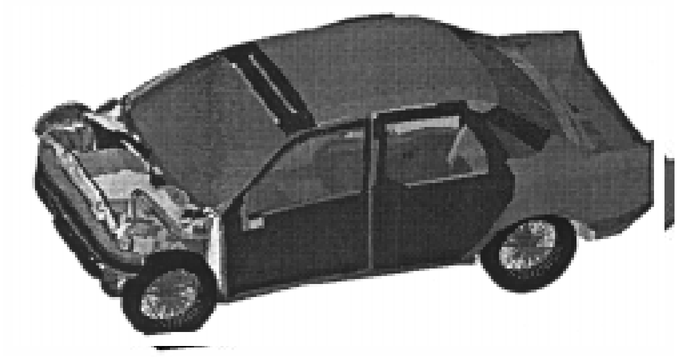


Figure 1.

VECTORIZATION

The initial investigation into FCRASH vectorization was performed by the University of Michigan's Advanced Computer Architecture Laboratory. A program should be optimized for single processor performance before it is parallelized. A high level of single processor vectorization is also a must for efficient utilization of the CRAY- C90 architecture. The Cray FOR-

TRAN optimization manual states that vectorization can yield execution up to 10 times faster than scalar processing.

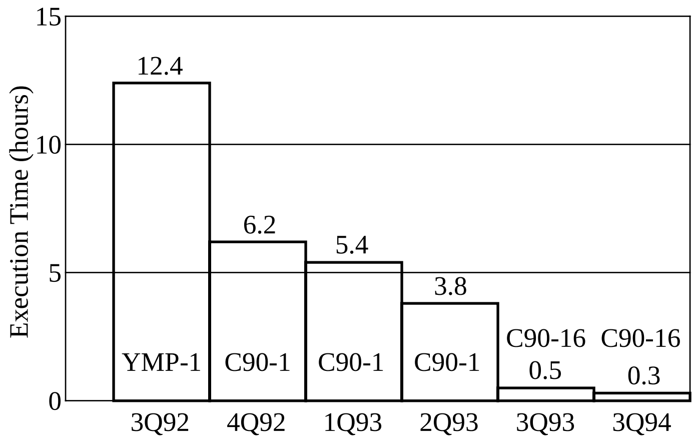


Figure 2.

The original version of FCRASH was written for a RISC workstation, thus there was no attempt to vectorize the element calculations. The typical code fragment took a single element and passed it sequentially through several different subroutines, each performing some part of the element force calculation on only a single element (see Program Example 1).

```
   REAL A(NUM_ELEMS), B(NUM_ELEMS), C(NUM_ELEMS)
   DO I=1, NUM_ELEMS
     CALL SOLVE (A(I), B(I), C(I))
   END DO
   END

   SUBROUTINE SOLVE(A, B, C)
   REAL A, B, C
* SCALAR
     A = FUN1(B) + FUN2(C)
   RETURN
   END
```

Program Example 1

Needless to say this yielded poor performance on a parallel vector processor (PVP) machine, such as the CRAY-C90.

The first phase of optimization was to replace these single element calls with a call to the subroutine with the entire list of elements to be processed (see Program Example 2). The list was then strip-mined into groups of elements compatible with the CRAY vector register length. The trade-off between vector lengths and its effect on parallel execution will be discussed later.

Further optimization included standard techniques such as subroutine inlining (both manually and automatically with the inline compiler option), loop unrolling, loop reordering, rewriting of scalar loops and the use of CRAY library subroutines where possible. Applying these techniques alone led to about 25% performance improvement.

The next step was to analyze the overall performance of the code. The performance monitoring tools available on the CRAY-C90 system were indicating excessive amount of data movement in relation to floating point calculations. By using the tool perfview[3], we were able to find the routines that were most affected by this factor.

```
   REAL A(NUM_ELEMS), B(NUM_ELEMS), C(NUM_ELEMS)
     CALL SOLVE (NUM_ELEMS, A, B, C)
   END

   SUBROUTINE SOLVE(NUM_ELEMS, A, B, C)
   REAL A(NUM_ELEMS), B(NUM_ELEMS), C(NUM_ELEMS)
* VECTOR
   DO I=1, NUM_ELEMS
     A(I) = FUN1(B(I)) + FUN2(C(I))
   END DO
   RETURN
   END
```

Program Example 2

```
   REAL A(NUM_ELEMS), B(NUM_ELEMS), C(NUM_ELEMS)
     CALL SUB1 (NUM_ELEMS, A, B)
     CALL SUB2 (NUM_ELEMS, B, C)
   END

   SUBROUTINE SUB1 (NUM_ELEMS, A, B)
   REAL A(NUM_ELEMS), B(NUM_ELEMS)
   DO I=1, NUM_ELEMS
    B(I) = FUN1(A(I))
   END DO
   END
   SUBROUTINE SUB2 (NUM_ELEMS, B, C)
   REAL B(NUM_ELEMS), C(NUM_ELEMS)
   DO I=1, NUM_ELEMS
    C(I) = FUN2(B(I))
   END DO
   END
```

Program Example 3

Typically an array of elements that needed to be solved was passed through several subroutines, with each subroutine adding only a small amount of additional computation, but loading and storing several intermediate arrays (Program Example 3).

```
   REAL A(NUM_ELEMS), C(NUM_ELEMS)
     CALL SUB (NUM_ELEMS, A, C)
   END

   SUBROUTINE SUB (NUM_ELEMS, A, C)
   REAL A(NUM_ELEMS), B, C(NUM_ELEMS)
   DO I=1, NUM_ELEMS
    B = FUN1 (A(I))
    C(I) = FUN2 (B)
   END DO
   END
```

Program Example 4

After several intermediate calculations were combined in the same do loop, temporary arrays were eliminated or replaced with scalar variables (Program Example 4). Merging of the subroutines had a secondary effect of removing the subroutine call overhead associated with the eliminated routines.

PARALLELIZATION

After the code had sufficient single processor performance the next task was to parallelize the code to achieve our goal of 30 minute turnaround time. The first task of this effort was to break up element force calculations to work on groups of elements to achieve the benefits of vectorization and provide the basis for parallel execution by distributing groups of elements to available processors. A parameter, MAXVL, was introduced to control the number of elements to be distributed to processors at a time (see Program Example 5).

```
   REAL A(NUM_ELEMS), B(NUM_ELEMS), C(NUM_ELEMS)
   PARAMETER (MAXVL=256)
* PARALLEL
   DO N1=1,NUM_ELEMS,MAXVL
     N2 = MIN(N1+MAXVL-1, NUM_ELEMS)
     CALL SOLVE (N2-N1+1, A(N1), B(N1), C(N1))
   END DO
   END

   SUBROUTINE SOLVE(LEN, A, B, C)
   PARAMETER (MAXVL=256)
   REAL A(MAXVL), B(MAXVL), C(MAXVL)
* VECTOR
   DO I=1, LEN
      A(I) = FUN1(B(I)) + FUN2(C(I))
   END DO
   RETURN
   END
```

Program Example 5

While optimizing the element calculations, we studied the effect of the group size (MAXVL) on the overall code performance. An experiment was performed varying the MAXVL parameter from 128 and to 1024. As expected the single processor performance improved as the vector length increased, with largest improvement achieved when MAXVL approached even multiples of 128, the vector pipe length of the hardware. Looking at only the even multiples of 128 vector lengths, the curve seem to flatten out starting at a vector length of 512 (see Figure 3). Even though the vector length of 512 improved the single processor performance by 9.6% over the vector length of 256, the parallel performance suffered by even larger margin due to decreasing number of available groups of elements (chunks of work) to be distributed to processors, leading to load balancing problems. Given a 22,000 element model, each processor of a 16-processor system gets about 5 chunks of works with MAXVL set to 256, or 2 chunks for MAXVL set to 512. By comparing the effects of MAXVL on performance on a single processor and in parallel, we settled on the vector length of 256 to maintain high degree of vectorization and sufficient number of groups of elements (chunks) to achieve reasonable load balancing in the parallel environment. Our plan is to revisit this in the future with larger models and better load balancing techniques.

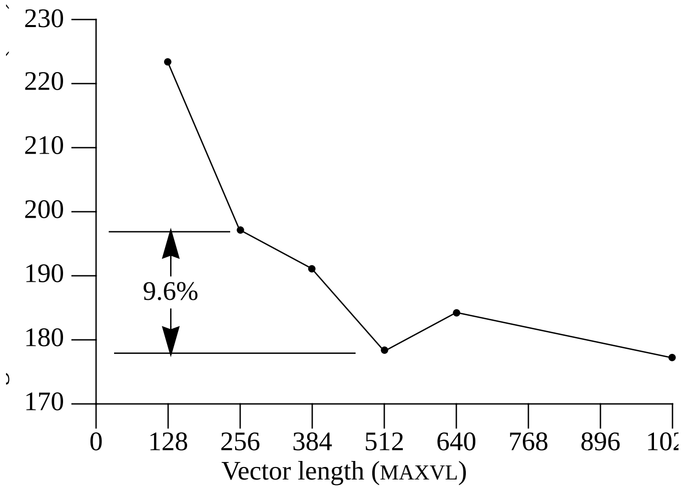


Figure 3.

Further efforts concentrated on achieving the maximum load balancing while maintaining the vector performance. Some of the techniques utilized in this effort were combining multiple parallel regions (one for each type element) into a single region covering all types of elements in the model and weighting the cost of various types of elements in the distribution mechanism.

Another parallel optimization performed was the merging of several independent blocks of code to be performed in parallel. The idea was to reduce the amount of parallel region overhead and get a better load balancing by dividing the number of processors into a much larger body of work. At the end of every time step several routines are called to advance the velocities, angular velocities, and energy calculations. All these routines which were being performed independently were merged into a single parallel region with the work distributed to processors in chunks of nodes, and all necessary calculations performed by each processor for its chunks.

The largest effort went into parallelizing the contact algorithm. The code was restructured to distribute both global and local searches of the algorithm. Again, the main difficulty was in achieving and maintaining a high degree of load balancing, which we overcame by parallelizing the contact search algorithm to work on chunks of nodes on a processor and developing a cost criteria to be used in the loop distribution algorithm.

I/O OPTIMIZATION

Output of data is a synchronization point in the program so all processors must wait for completion of I/O before proceeding with their work. Although overall cost of I/O in this code is fairly small (about 3% of the execution time on a single processor), it greatly affected overall speedup during parallel execution. We were able to cut the cost of I/O in half by applying standard I/O optimization techniques such as rewriting multiple WRITE statements within explicit do loops into single WRITE statements with implied do loops (see Program Example 6), and combining multiple WRITE statements into one.

Analysis of the FCRASH output showed that substantial amount of data was written for validation purposes during development only and was not necessary in production environment. Eliminating unnecessary output further reduced the time spent in I/O.

```
*ORIGINAL CODE
   DO I=1, NUM_NODES
    WRITE (unit,10) X(I), Y(I), Z(I)
   END DO

*CHANGED CODE
   WRITE (unit,10) (X(I),Y(I),Z(I),I=1,NUM_NODES)
```

Program Example 6

Binary I/O was also explored and used for some timing estimates, but not implemented in the production code due to Cray non-standard binary format and incompatibility with post-processing software.

IDLE CPU HOLD TIME

While running our benchmark models, we studied the effects of varying the length of time a processor holds on to an idle cpu before releasing it, the MP_HOLDTIME parameter on the program execution time. We were able to vary the execution time by as much as 10% and currently exploring optimal use of this parameter in a production environment.

RESULTS AND FUTURE GOALS

At the end of this effort our performance goal was exceeded by 33%. A number of larger models were also benchmarked which achieved a speedup of 9.5 to 10 on the 16 processor CRAY-C90 system. The code has also been executed on the new CRAY-J90 and CRAY-T90 systems and showed excellent performance. Future projects include the tuning of the PVP version of FCRASH to take full advantage of the new Cray systems, and to explore more creative ways to increase the program speedup with multiple processors. To gain an even higher level of performance for this program new types of architecture are being explored, namely; the CRAY-T3D, the IBM-SP2, and the Convex SPP. The code is currently being ported to the CRAY-T3D architecture by members of the FCRASH group in conjunction with researchers at the University of Michigan's Advanced Computer Architecture Laboratory and the University of Michigan Center for Parallel Computing.

REFERENCES

[1] C. H. Farhat and L. Crivelli, A General Approach to Nonlinear FE Computations on Shared Memory Multi-Processors, Rep. No. CU-CSSC-87-09, University of Colorado, Boulder, CO, 1987.
[2] K. Hwang, F. A. Briggs, Computer Architecture and Parallel Processing. McGraw-Hill, 1984.
[3] Cray Research Inc., UNICOS Performance Utilities Reference Manual, SR-2040 7.0, May 1992.
[4] C. H. Farhat, E. Wilson, and G. Powell, Solutions of Finite Element Systems on Concurrent Processing Computers, Engineering Computing, Vol 2, pp 157-165, 1987.
[5] J. G. Malone, Parallel Nonlinear Dynamic Finite Element Analysis of Three-Dimensional Shell Structures, Computers & Structures, Vol. 35, No. 5, pp. 523-539, 1990.■