Aerospace Software Final

Thanh Cong Bui[*], Tucker Emmett[†]

ASEN 4057

I. Description of C Program to Improve

The Apollo 13 Problem code was selected to improve upon with changes applied to both the integration scheme and an improvement in runtime through parallel processing. A brief overview of the problem follows below.

NASA's Apollo 13 mission was intended to land on the moon. Unfortunately, an onboard explosion demanded a speedy and safe return of the astronauts from near-Lunar orbit back to Earth. This project explores the numerical optimization necessary to quickly calculate a return trajectory for the Apollo 13 module such that the spacecraft safely returns to Earth.

The mathematics regarding this simulation are straightforward. This problem is treated as a two-dimensional two-body problem, with the Apollo spacecraft and the Moon as the two moving bodies. This report will not include a discussion of the differential equations driving the problem, as gravitational fields are the only elements in this case.

The preliminary Apollo 13 code was developed in MATLAB, which offered a robust minimization function to determine minima in both time to return to Earth and velocity change (delta V). Subsequent software developments extended into C, which made use of a Runge-Kutta 4th order solver to perform numerical integrations. These numerical integrations were combined with a straightforward optimization sweep over a wide range of velocities to determine optimal values.

The preliminary C Code has a couple of components that will be addressed by subsequent improvements. These components are discussed below.

- The C Code is entirely serial. The code is flop-intensive, as it must search through a wide range of velocities at a very small velocity increment to determine an optimal solution.
- The C Code uses a Runge-Kutta 4th Order integration scheme. While a 4th order solver is more accurate than a coarse Euler's Method, the 4th Order scheme uses a fixed timestep.

Proposed solutions to these problems follow below.

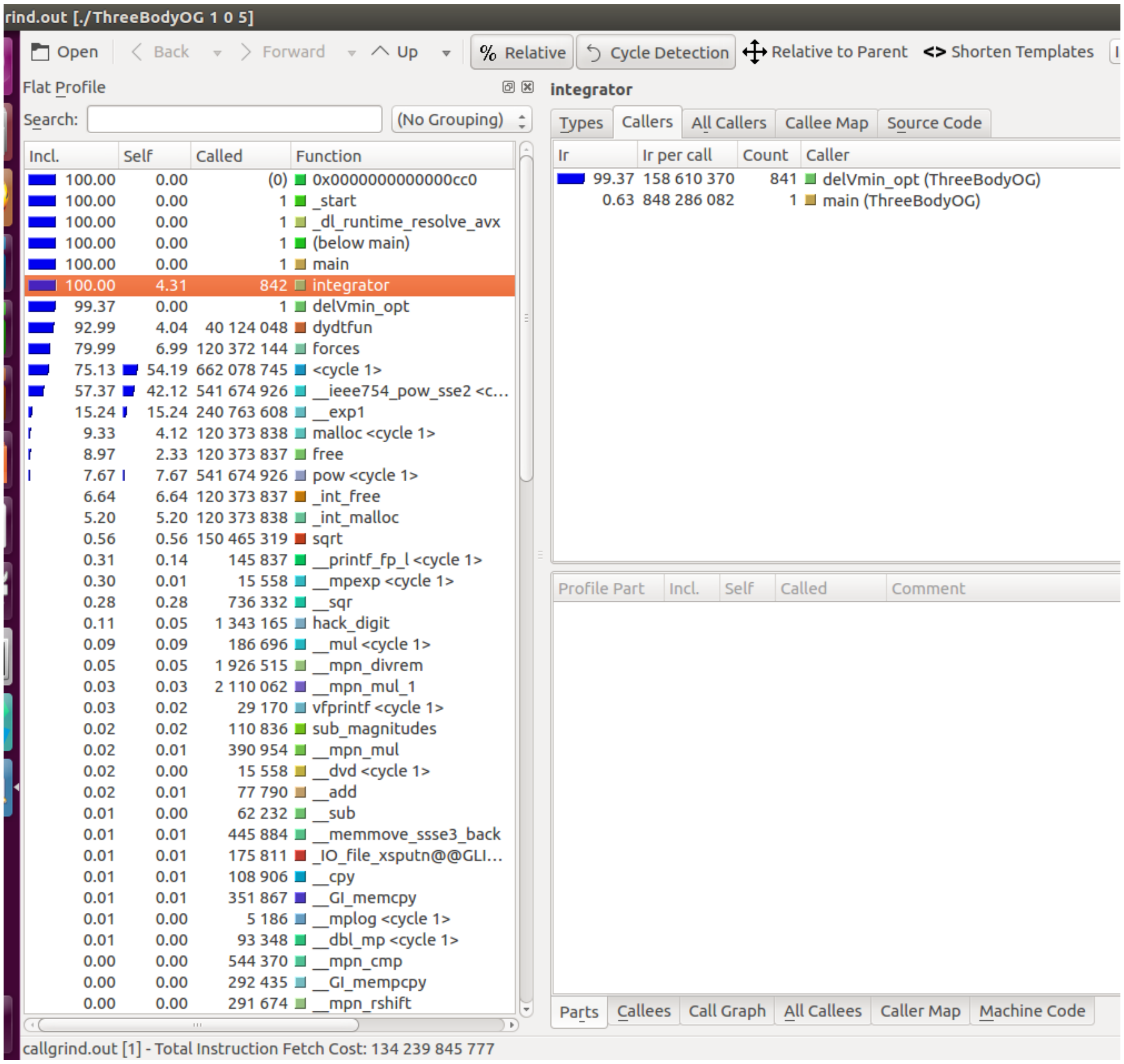
- Parallelize the C Code. As most computers have multiple cores and processors, it is possible to divide flops among multiple processors to reduce runtime.
- Use a Runge-Kutta 4th-5th Order integration scheme. This scheme will make use of a variable timestep to allow for more coarse timesteps when the problem's derivatives are small, and more fine timesteps when the derivatives are very large.

The C Code currently consists of 6 files. The main file *main.c* consists of declarations to pipe instructions to corresponding optimization schemes. At the culmination of these schemes, *main.c* creates output files and organizes the optimization values for the user to analyze. The Runge-Kutta 4th-5th Order integration scheme, *integrator.c*, consists of the entire method necessary to perform an accurate variable timestep integration. The force calculation file *forces.c* calculates the gravitational force acting on the satellite and Moon for a given distance. The optimization schemes are contained within files *delVminopt.c* and *delVtimeopt.c*, which optimize for a minimum velocity and return time, respectively. Finally, the file *myrename.c* is included for completion.

[*]SID:104537876
[†]SID: 101514145

II. Part 1: Serial Code Optimization

A. Profile Report and Discussion of Performance of Original C Code



Incl.	Self	Called	Function
100.00	0.00	(0)	0x0000000000000cc0
100.00	0.00	1	_start
100.00	0.00	1	_dl_runtime_resolve_avx
100.00	0.00	1	(below main)
100.00	0.00	1	main
100.00	4.31	842	integrator
99.37	0.00	1	delVmin_opt
92.99	4.04	40 124 048	dydtfun
79.99	6.99	120 372 144	forces
75.13	54.19	662 078 745	<cycle 1>
57.37	42.12	541 674 926	__ieee754_pow_sse2 <c…
15.24	15.24	240 763 608	__exp1
9.33	4.12	120 373 838	malloc <cycle 1>
8.97	2.33	120 373 837	free
7.67	7.67	541 674 926	pow <cycle 1>
6.64	6.64	120 373 837	_int_free
5.20	5.20	120 373 838	_int_malloc
0.56	0.56	150 465 319	sqrt
0.31	0.14	145 837	__printf_fp_l <cycle 1>
0.30	0.01	15 558	__mpexp <cycle 1>
0.28	0.28	736 332	__sqr
0.11	0.05	1 343 165	hack_digit
0.09	0.09	186 696	__mul <cycle 1>
0.05	0.05	1 926 515	__mpn_divrem
0.03	0.03	2 110 062	__mpn_mul_1
0.03	0.02	29 170	vfprintf <cycle 1>
0.02	0.02	110 836	sub_magnitudes
0.02	0.01	390 954	__mpn_mul
0.02	0.00	15 558	__dvd <cycle 1>
0.02	0.01	77 790	__add
0.01	0.00	62 232	__sub
0.01	0.01	445 884	__memmove_ssse3_back
0.01	0.01	175 811	_IO_file_xsputn@@GLI…
0.01	0.01	108 906	__cpy
0.01	0.01	351 867	__GI_memcpy
0.01	0.00	5 186	__mplog <cycle 1>
0.01	0.00	93 348	__dbl_mp <cycle 1>
0.00	0.00	544 370	__mpn_cmp
0.00	0.00	292 435	__GI_mempcpy
0.00	0.00	291 674	__mpn_rshift

Ir	Ir per call	Count	Caller
99.37	158 610 370	841	delVmin_opt (ThreeBodyOG)
0.63	848 286 082	1	main (ThreeBodyOG)

Profile Part	Incl.	Self	Called	Comment

callgrind.out [1] - Total Instruction Fetch Cost: 134 239 845 777

Figure 1. Profile of Original Code

The original code called the function dydtfun() for 40 million times. This function is the main one that does computations. This is where we can go to optimize code, because even though the function forces() is called more, it is already short and efficient. We can also try to minimized the number of calls for dydtfun().

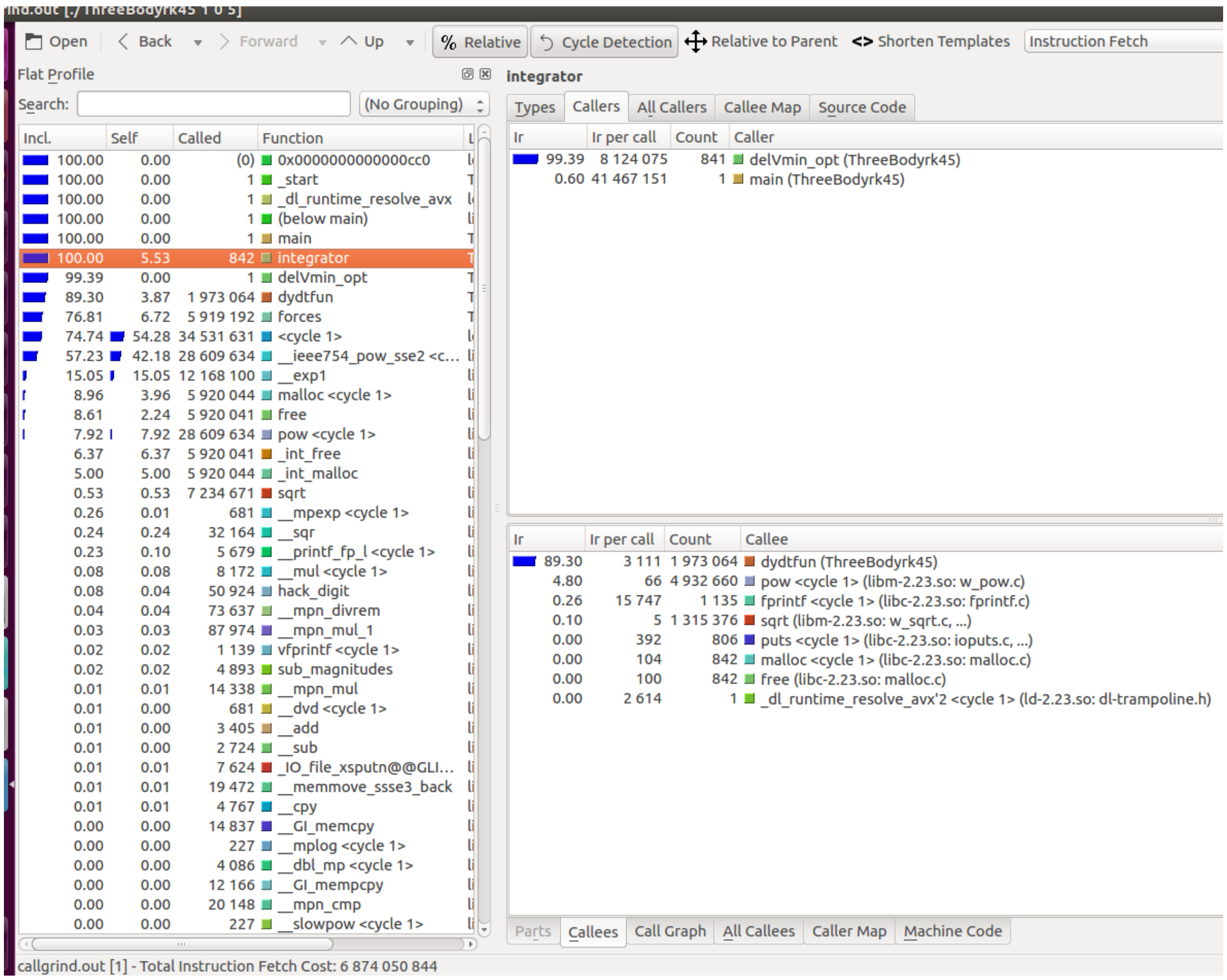
B. Proposed Serial Code Improvements

The serial code can be improved by implementation of an adaptive time-integration scheme such as RK45. The function integrator() has been modified to estimate the slope of the ODE functions at 6 different steps. Then, the Runge-Kutta 4th order approximation was compared with the 5th order approximation to find the approximation error. If this error was bigger than a specified tolerance, then the time step h would be increased or decreased in a manner inversely proportional to the error. This allows us to take a bigger time step when the trajectory is not changing too drastically, and ultimately decrease the operation count of the code.

C. Timing Comparison of Original C Code and Optimized C Code

Each version of code was ran for the arguments:
Objective 1
Clearance = 0m
Accuracy = 5m
With original code, time spent: 32.85 seconds
With RK45 implementation, time spent : 1.73seconds
We can see here that the time for the RK45 implementation is much less. It is about 20 times faster. We can see that the adaptive time-integration scheme makes the code much more efficient.

D. Profile Report and Discussion of Performance of Optimized C Code



ind.out [./ThreeBodyrk45 1 0 5]

Incl.	Self	Called	Function
100.00	0.00	(0)	0x0000000000000cc0
100.00	0.00	1	_start
100.00	0.00	1	_dl_runtime_resolve_avx
100.00	0.00	1	(below main)
100.00	0.00	1	main
100.00	5.53	842	integrator
99.39	0.00	1	delVmin_opt
89.30	3.87	1 973 064	dydtfun
76.81	6.72	5 919 192	forces
74.74	54.28	34 531 631	<cycle 1>
57.23	42.18	28 609 634	__ieee754_pow_sse2 <c...
15.05	15.05	12 168 100	__exp1
8.96	3.96	5 920 044	malloc <cycle 1>
8.61	2.24	5 920 041	free
7.92	7.92	28 609 634	pow <cycle 1>
6.37	6.37	5 920 041	_int_free
5.00	5.00	5 920 044	_int_malloc
0.53	0.53	7 234 671	sqrt
0.26	0.01	681	__mpexp <cycle 1>
0.24	0.24	32 164	__sqr
0.23	0.10	5 679	__printf_fp_l <cycle 1>
0.08	0.08	8 172	__mul <cycle 1>
0.08	0.04	50 924	hack_digit
0.04	0.04	73 637	__mpn_divrem
0.03	0.03	87 974	__mpn_mul_1
0.02	0.02	1 139	vfprintf <cycle 1>
0.02	0.02	4 893	sub_magnitudes
0.01	0.01	14 338	__mpn_mul
0.01	0.00	681	__dvd <cycle 1>
0.01	0.00	3 405	__add
0.01	0.00	2 724	__sub
0.01	0.01	7 624	_IO_file_xsputn@@GLI...
0.01	0.01	19 472	__memmove_ssse3_back
0.01	0.01	4 767	__cpy
0.00	0.00	14 837	__GI_memcpy
0.00	0.00	227	__mplog <cycle 1>
0.00	0.00	4 086	__dbl_mp <cycle 1>
0.00	0.00	12 166	__GI_mempcpy
0.00	0.00	20 148	__mpn_cmp
0.00	0.00	227	__slowpow <cycle 1>

callgrind.out [1] - Total Instruction Fetch Cost: 6 874 050 844

integrator

Ir	Ir per call	Count	Caller
99.39	8 124 075	841	delVmin_opt (ThreeBodyrk45)
0.60	41 467 151	1	main (ThreeBodyrk45)

Ir	Ir per call	Count	Callee
89.30	3 111	1 973 064	dydtfun (ThreeBodyrk45)
4.80	66	4 932 660	pow <cycle 1> (libm-2.23.so: w_pow.c)
0.26	15 747	1 135	fprintf <cycle 1> (libc-2.23.so: fprintf.c)
0.10	5	1 315 376	sqrt (libm-2.23.so: w_sqrt.c, ...)
0.00	392	806	puts <cycle 1> (libc-2.23.so: ioputs.c, ...)
0.00	104	842	malloc <cycle 1> (libc-2.23.so: malloc.c)
0.00	100	842	free (libc-2.23.so: malloc.c)
0.00	2 614	1	_dl_runtime_resolve_avx'2 <cycle 1> (ld-2.23.so: dl-trampoline.h)

Figure 2. Profile of Optimized Code

Here we see that the calls to the dydtfun() function is reduced to 2 million. This is about a 20 times reduction, which is probably why the runtime also decreased by about 20 times.

III. Part 2: Parallelization of Optimized Code

A. Proposed Strategy for Code Parallelization

The minimum velocity routine will be parallelized by this code. The minimum time routine parallelization would follow a similar schematic. The parallelization strategy logic follows below:

- Parallelize the search strategy. Make each process search through a specific range of delta V's.
- Create a variable *increment*. This variable will be used as a "meshing" variable to split the work across each process. *Increment* is equal to the magnitude of the delta V search range divided by the number of processes. For a search range from -10 to 10 m/s utilizing 4 processes, *increment* takes the value of $20/5 = 4$.
- Create the variable *startValue*. This value changes based on the rank of the reference process. For the 0^{th} process, *startValue* is equal to the low end of the search range. For the $n-1^{th}$ process, *startValue* is equal to the upper end minus *increment*. It is through this logic that the computation is "meshed" across the available processes.
- Create the variable *endValue*. This value changes based on the rank of the reference process. For the 0^{th} process, *endValue* is equal to the low end of the search range plus *increment*. For the $n-1^{th}$ process, *endValue* is equal to the upper end of the search range.
- Iterate across the size of the search range, allowing the code to direct the computations to the proper processors.
- Use MPI-Gather to collect the optimal values calculated from each process.
- Design a simple bit of logic to select the smallest optimal value as the true optimal value over all the processes.

B. Scalability Study on Virtual Machine with up to Eight Processes

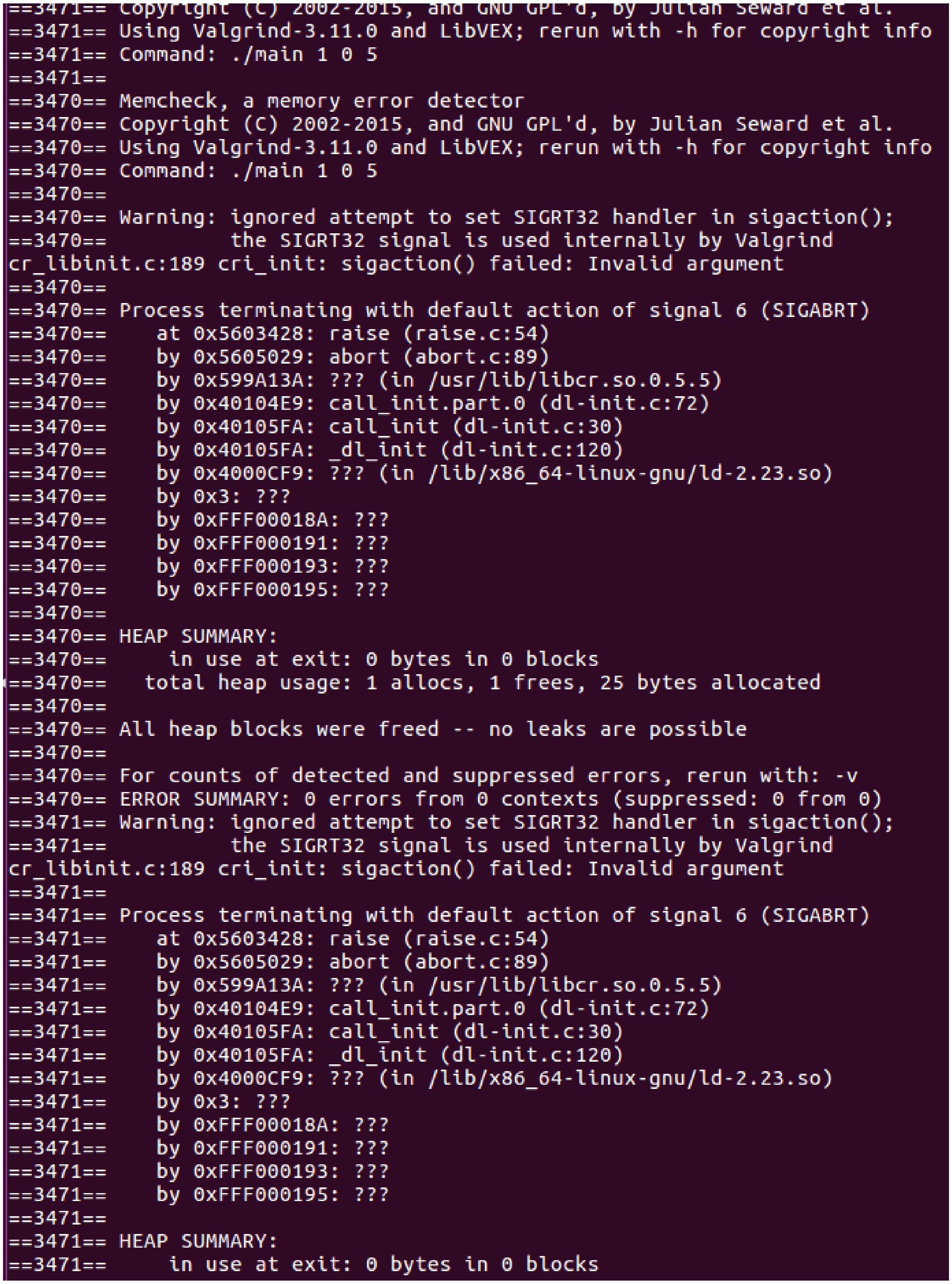
Unfortunately, this code appears to scale poorly with an increasing amount of processes. A list of runtimes with corresponding process numbers has been included for comparison. For reference, the C Code written in serial takes 54.446 seconds to return an answer within a small desired tolerance. All times listed below are in seconds, with command line input values of *1 0 5*.

N. Processes	Execution Time (s)
1	0.0752
2	0.1154
3	0.2181
4	0.2258
5	0.4029
6	0.2134
7	0.7615
8	0.5450

It is interesting to note that an even number of processes almost always results in a faster and more accurate solution. This is likely due to the solution sensitivity to an initial guess, as well as the "meshing" scheme used to parallelize the search grid. It is speculated that the solution lies in a location which, with an odd number of processes, can be mistakenly skipped during iteration with variable accuracy. For an unknown reason, an even number of processes appears to properly iterate over this region and produce the desired solution of a total delta V in the range of [0, 49] m/s. It appears that, with increasing processes, the communication time required to gather the optimums from each process starts to bog down the code. The software developers speculate that because the optimization routine is not overly complex, a two-process parallel algorithm performs quite well. If the code was more complicated or involved a detailed optimization search method, multiple processes would likely perform better than just two.

C. Profile Report and Discussion of Performance on Virtual Machine

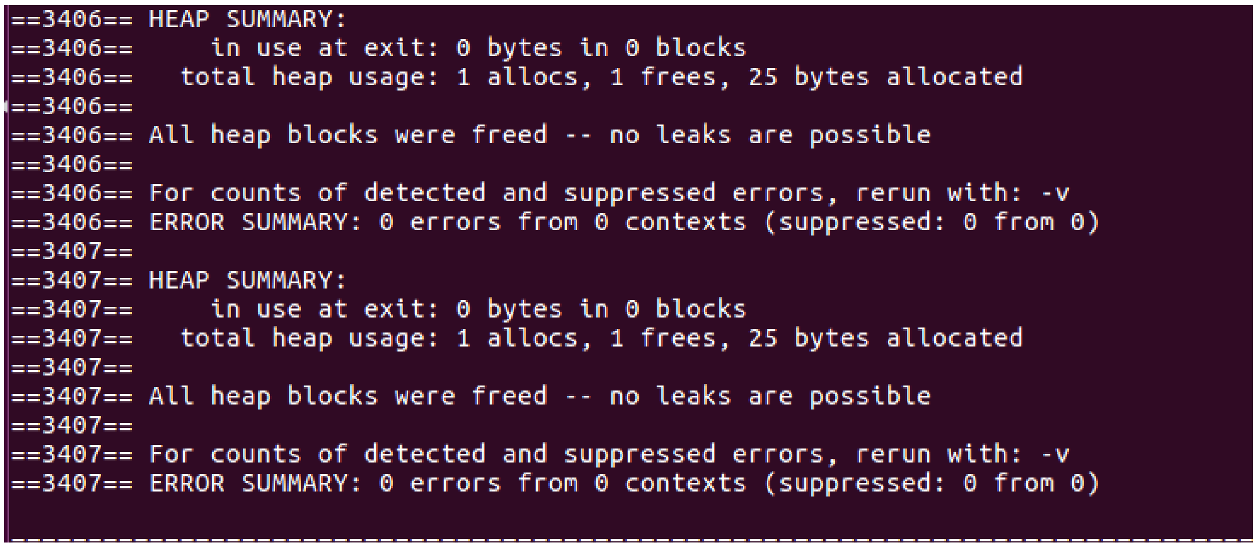
Profiling this parallel code produces the following command line output. At this time it appears that there is an error with the callgrind functionality, so a simple valgrind call has been used instead.



```
==3471== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3471== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3471== Command: ./main 1 0 5
==3471==
==3470== Memcheck, a memory error detector
==3470== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3470== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3470== Command: ./main 1 0 5
==3470==
==3470== Warning: ignored attempt to set SIGRT32 handler in sigaction();
==3470==          the SIGRT32 signal is used internally by Valgrind
cr_libinit.c:189 cri_init: sigaction() failed: Invalid argument
==3470==
==3470== Process terminating with default action of signal 6 (SIGABRT)
==3470==    at 0x5603428: raise (raise.c:54)
==3470==    by 0x5605029: abort (abort.c:89)
==3470==    by 0x599A13A: ??? (in /usr/lib/libcr.so.0.5.5)
==3470==    by 0x40104E9: call_init.part.0 (dl-init.c:72)
==3470==    by 0x40105FA: call_init (dl-init.c:30)
==3470==    by 0x40105FA: _dl_init (dl-init.c:120)
==3470==    by 0x4000CF9: ??? (in /lib/x86_64-linux-gnu/ld-2.23.so)
==3470==    by 0x3: ???
==3470==    by 0xFFF00018A: ???
==3470==    by 0xFFF000191: ???
==3470==    by 0xFFF000193: ???
==3470==    by 0xFFF000195: ???
==3470==
==3470== HEAP SUMMARY:
==3470==     in use at exit: 0 bytes in 0 blocks
==3470==   total heap usage: 1 allocs, 1 frees, 25 bytes allocated
==3470==
==3470== All heap blocks were freed -- no leaks are possible
==3470==
==3470== For counts of detected and suppressed errors, rerun with: -v
==3470== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==3471== Warning: ignored attempt to set SIGRT32 handler in sigaction();
==3471==          the SIGRT32 signal is used internally by Valgrind
cr_libinit.c:189 cri_init: sigaction() failed: Invalid argument
==3471==
==3471== Process terminating with default action of signal 6 (SIGABRT)
==3471==    at 0x5603428: raise (raise.c:54)
==3471==    by 0x5605029: abort (abort.c:89)
==3471==    by 0x599A13A: ??? (in /usr/lib/libcr.so.0.5.5)
==3471==    by 0x40104E9: call_init.part.0 (dl-init.c:72)
==3471==    by 0x40105FA: call_init (dl-init.c:30)
==3471==    by 0x40105FA: _dl_init (dl-init.c:120)
==3471==    by 0x4000CF9: ??? (in /lib/x86_64-linux-gnu/ld-2.23.so)
==3471==    by 0x3: ???
==3471==    by 0xFFF00018A: ???
==3471==    by 0xFFF000191: ???
==3471==    by 0xFFF000193: ???
==3471==    by 0xFFF000195: ???
==3471==
==3471== HEAP SUMMARY:
==3471==     in use at exit: 0 bytes in 0 blocks
```

Figure 3. Profile of Parallel Code

A memory check of the software follows below. Fortunately, the check reports that there are no memory leaks. The usage of free() within the software was likely useful in avoiding memory leaks.



```
==3406== HEAP SUMMARY:
==3406==     in use at exit: 0 bytes in 0 blocks
==3406==   total heap usage: 1 allocs, 1 frees, 25 bytes allocated
==3406==
==3406== All heap blocks were freed -- no leaks are possible
==3406==
==3406== For counts of detected and suppressed errors, rerun with: -v
==3406== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
==3407==
==3407== HEAP SUMMARY:
==3407==     in use at exit: 0 bytes in 0 blocks
==3407==   total heap usage: 1 allocs, 1 frees, 25 bytes allocated
==3407==
==3407== All heap blocks were freed -- no leaks are possible
==3407==
==3407== For counts of detected and suppressed errors, rerun with: -v
==3407== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figure 4. Memory Check of Parallel Code

Overall, the parallel code exhibits no memory leaks and appears to be very clean and well-behaved in nature.

IV. Summary of Findings and Potential Futures Improvements

It was found during the code parallelization process that creating efficient, well-written parallel code can be tremendously difficult. Parallelization offered enormous time improvements over the serial process–a minimal speedup of 71x–and was found to be a very useful tool in advanced software development. Future improvements will likely come in the form of advancements in the optimization scheme. There are certainly better optimization tools and packages which offer a refined approach; the scope of this project allowed for little more than a simple sweep over a known range of values. It is known that the optimization scheme experienced significant speedups with the use of parallel processes; perhaps with a little more time, the code might be parallelized and made more efficient so that the even/odd process peculiarities do not show up. One significant improvement would be one to remove the even/odd process behavior. As can be seen in above tables, adding more processes actually *slowed down* the parallel code in some cases. This is a peculiar characteristic, and one that is certainly worth exploring further.