



Programming in the HPC Environment

Dr. Stefan Albensoeder

Contact: Stefan.Albensoeder@uni-oldenburg.de

Overview

1. Introduction
2. Compiler
3. Debugging tools
4. Profiling
5. Optimization

Available development tools

Compiler (C, C++, Fortran)

- Intel Cluster Studio
- GNU compiler
- PGI Accelerator Suite
- Open64
- CLang

MPI

- OpenMPI
- Intel MPI

Libraries

- BLAS/LAPACK in MKL
- NAG
- LEDA
- FFTW
- NetCDF, HDF5
- ...

Other languages

- Python, R
- ncarg/ncl, cdo
- Matlab, Octave

COMPILER

GNU Compiler

Languages/command

- C gcc
- C++ g++
- Fortran gfortran

Advantages

- high availability, free
- new standards implemented
- fast compilation
- performance of resulting C++ usually good
- most open source software tested with GNU

Recommended module

- gcc/4.8.1

Disadvantages

- performance of not optimized C and Fortran code moderate

Intel Compiler (from Intel Cluster Studio)

Languages/command

- C `icc`
- C++ `icpc`
- Fortran `ifort`

Recommended module

- `ics/2013_sp1.3.174/64`

Advantages

- usually best performance for C and Fortran code on Intel processors
- high performance with OpenMP
- high optimization by compiler
- Fortran extension `coarray` implemented

Disadvantages

- performance of C++ code can be moderate
- slow compilation
- not free, only for Intel architecture available
- newest standards may missing
- sometimes too enthusiastic in optimization

PGI Compiler (from PGI Accelerator Suite)

Languages/command

- C `pgcc`
- C++ `pgCC`
- Fortran `pgfortran`
 `(pgf77, ...)`

Recommended module

- `pgi/13.10`

Advantages

- good performance for C, C++ and Fortran code
- OpenACC is implemented (extension for simple usage of accelerators like GPUs)
- Fortran extension coarray implemented

Disadvantages

- slow compilation
- not free
- newest standards may missing
- OpenACC not a common standard (probably implemented in gcc 5.x)

Open64

Languages/command

- C `opencc`
- C++ `openCC`
- Fortran `openf90,`
`openf95`

Advantages

- open source
- good performance for C, C++ and Fortran code (e.g. on AMD processors)
- available for several architectures

Recommended module

- `open64/4.5.2.1`

Disadvantages

- not commonly used
- several branches exist

Clang

Languages/command

- C clang
- C++ clang++
- Fortran -

Advantages

- open source
- fast compilation
- good code check
- available for several architectures

Recommended module

- clang/3.5.0

Disadvantages

- moderate performance
- no Fortran compiler
- support for OpenMP up to now not standard

Compilation

General steps

- before compilation load compiler module (Intel Compiler for C, C++ and Fortran, GNU for C++)
- recommended: create (modify) `Makefile`
 - compiler commands
 - compiler optimization settings (shown later)
- compile your code, e.g. by command `make`

Hints

- enable all warnings (GNU: `-Wall`, Intel: `-w3`)
- different compilers, different warnings
→ compile code with different compilers
- try to remove all warnings
→ can avoid bugs

DEBUGGING

Debugging

- there exists more tools for debugging than print-statements in the code
 - debugger on FLOW/HERO (interactive stepping through code, monitor/manipulate variables,...)
 - `gdb` (command line, GNU, GUI frontend `ddd`)
 - `idb` (GUI, Intel Cluster Studio)
 - `pgdbg` (GUI, PGI Accelerator Studio)
 - debugging toolset `valgrind`
 - `memcheck` (memory leaks, usage of uninitialized memory,...)
 - `callgrind` (call graph analyzer)
 - `cachegrind` (cache profiler)
 - `hellgrind` (race conditions in multithreaded code)
 - `massif` (heap profiler)
 - ...

Debugging

- multithreaded code
 - valgrind tool hellgrind (race conditions in multithreaded code)
 - Intel Inspector `inspxe-gui` (memory and thread checker)
- MPI
 - Intel Trace Analyser ITAC

Debugging

Preparations for all debugging/profiling tools

- include debugging information
 - enable source code/line number information
 - for all compiler: add option `-g` to compiler/linker flags
 - maybe reduce optimization levels and use GNU compiler
 - recompile the complete code (not only linking)
- if possible create simple case
 - small memory requirements
(memory requirements of tools can be high, especially if using memory checker)
 - low wall time consumption
(code execution in debugger 10-100 times slower)

DEBUGGING

GDB / DDD

Debugging – gdb/ddd

small C-test program

```
17 int main(int argc, char *argv[])
18 {
19     double *array;
20     int N = 10000;
21     int i;
22
23     /* init array */
24     for(i=0; i<=N; i++)
25         array[i] = 0.;
26
27     /* ..... */
28
29     return 0;
30 }
```

```
flow01> ./debugTest
Segmentation fault
```


Debugging – gdb/ddd

How to start to debug a code

- recompile program with compiler option `-g`
- run `ddd`
- load executable (program)
- click `<run>`
- see where debugger stops
- maybe analyze variables

Debugging – gdb/ddd

```
DDD: /user/fw/dihl9738/Lehre/HPC/2015/Programs/ProgrammingHPC/debugTest/debugTest.c
File Edit View Program Commands Status Source Data Help
(): main
Lookupt Find>> Break Watch Print Display Plot Show Rotate Set Undo

***** /
/*-----
| global include files
-----*/
#include <stdio.h>
#include <stdlib.h>

/*****
* main routine
*****/
int main(int argc, char *argv[])
{
    int N = 10000;
    double *array;
    int i;

    /* init array */
    for(i=0; i<=N; i++)
    array[i] = 0.;

    /* ..... */

Program received signal SIGSEGV, Segmentation fault.
0x0000000004004a1 in main (argc=1, argv=0x7fffffffdd88) at debugTest.c:25
Missing separate debuginfos, use: debuginfo-install
glibc-2.12-1.132.e16_5.2.x86_64 glibc-2.12-1.149.e16.x86_64
(gdb) ]

Program received signal SIGSEGV, Segmentation fault.
```

Debugging – gdb/ddd

correct C-test program

```
17 int main(int argc, char *argv[])
18 {
19     int N = 10000;
20     double *array = malloc((N+1) * sizeof(double));
21     int i;
22
23     /* init array */
24     for(i=0; i<=N; i++)
25         array[i] = 0.;
26
27     /* ..... */
28
29     return 0;
30 }
```

DEBUGGING VALGRIND MEMCHECKER

Debugging - valgrind

small C-test program

```
17 int main(int argc, char *argv[])
18 {
19     double *array = NULL;
20     int N = 10000;
21     int i;
22
23     /* allocate array */
24     array = malloc(N);
25
26     /* init array */
27     for(i=0; i<=N; i++)
28         array[i] = 0.;
29
30     /* ..... */
31
32     return 0;
33 }
```

Debugging - valgrind

```
flow01> valgrind --tool=memcheck ./valgrindTest
...
==14085== Invalid write of size 8
==14085==    at 0x4004E3: main (valgrindTest:28)
==14085==    Address 0x4c28750 is 0 bytes after a block of size 10,000 alloc'd
==14085==    at 0x4A088A4: malloc (vg_replace_malloc.c:291)
==14085==    by 0x4004C0: main (valgrindTest.c:24)
==14085==
==14085==
==14085== HEAP SUMMARY:
==14085==    in use at exit: 10,000 bytes in 1 blocks
==14085==    total heap usage: 1 allocs, 0 frees, 10,000 bytes allocated
==14085==
==14085== LEAK SUMMARY:
==14085==    definitely lost: 10,000 bytes in 1 blocks
==14085==    indirectly lost: 0 bytes in 0 blocks
==14085==    possibly lost: 0 bytes in 0 blocks
==14085==    still reachable: 0 bytes in 0 blocks
==14085==    suppressed: 0 bytes in 0 blocks
==14085== Rerun with --leak-check=full to see details of leaked memory
==14085==
==14085== For counts of detected and suppressed errors, rerun with: -v
==14085== ERROR SUMMARY: 8751 errors from 1 contexts (suppressed: 4 from 4)
```

Debugging - valgrind

small C-test program (**lines with problems**)

```
17 int main(int argc, char *argv[])
18 {
19     double *array = NULL;
20     int N = 10000;
21     int i;
22
23     /* allocate array */
24     array = malloc(N);
25
26     /* init array */
27     for(i=0; i<=N; i++)
28         array[i] = 0.;
29
30     /* ..... */
31
32     return 0;
33 }
```

Debugging - valgrind

correct C-test program

```
17 int main(int argc, char *argv[])
18 {
19     double *array = NULL;
20     int N = 10000;
21     int i;
22
23     /* allocate array */
24     array = malloc(N*sizeof(double));
25
26     /* init array */
27     for(i=0; i<N; i++)
28         array[i] = 0.;
29
30     /* ..... */
31     free(array);
32     return 0;
33 }
```


PROFILING

Profiling

Introduction

- profiling: dynamic program analysis that measures
 - function calls
 - duration of function calls
- information useful for systematic optimization
 - find out which function needs most of the time
 - spend time to optimize these functions

Preparations for profiling tools (without code modifications)

- add `-g` as compiler/linker flag (debugging symbols)
- add `-pg` as compiler/linker flag to instrument executable for profiling (Slows down the code. Use it only for profiling!)

Profiling

Profiling tools

- `gprof` (GNU, text output of the profiling report)
- `pgprof` (PGI Accelerator, GUI based)
- `amplxe-gui` (Intel Amplifier, GUI based, usable for multithreaded code)
- not shown: library calls within code

Profiling with `gprof`

- run executable as usual
 - during execution the file `gmon.out` will be created
- report by
`gprof nameOfExecutable`

Profiling - gprof

Example: Unknown source code which has to be optimized

```
flow01> make
gcc -O0 -g -pg -c profileTest.c
gcc -O0 -g -pg -c subroutines.c
gcc -g -pg profileTest.o subroutines.o -lm -o profileTest
flow01> ./profileTest
|A.x| = 9.98921e+07
flow01> ls
Makefile gmon.out profileTest profileTest.c profileTest.o subroutines.c
subroutines.h subroutines.o
flow01> gprof profileTest
Flat profile:
```

Each sample counts as 0.01 seconds.

%	cumulative	self		self	total	
time	seconds	seconds	calls	s/call	s/call	name
49.20	2.99	2.99	1	2.99	2.99	initMatrix
39.92	5.41	2.42	1	2.42	2.42	matVecMul
11.35	6.10	0.69	3	0.23	0.23	zeroVector
0.00	6.10	0.00	1	0.00	0.00	initVector
0.00	6.10	0.00	1	0.00	0.00	vecNorm
...						

Profiling - gprof

Call graph (explanation follows)

index	% time	self	children	called	name
					<spontaneous>
[1]	100.0	0.00	6.10		main [1]
		2.99	0.00	1/1	initMatrix [2]
		2.42	0.00	1/1	matVecMul [3]
		0.69	0.00	3/3	zeroVector [4]
		0.00	0.00	1/1	initVector [5]
		0.00	0.00	1/1	vecNorm [6]

[2]	49.0	2.99	0.00	1/1	main [1]
		2.99	0.00	1	initMatrix [2]

[3]	39.7	2.42	0.00	1/1	main [1]
		2.42	0.00	1	matVecMul [3]

[4]	11.3	0.69	0.00	3/3	main [1]
		0.69	0.00	3	zeroVector [4]

[5]	0.0	0.00	0.00	1/1	main [1]
		0.00	0.00	1	initVector [5]

[6]	0.0	0.00	0.00	1/1	main [1]
		0.00	0.00	1	vecNorm [6]

Profiling

First analysis

- 2 functions needs 90% of time
 - `initMatrix`
 - `matVecMul`
- number of function calls is not the problem
- program is used frequently with larger resolution
→ efforts in optimization will pay off

Note

- distribution of wall time could be problem dependent
→ use realistic cases for profiling

OPTIMIZATION

Motivation

Why spend time in optimization?

- solving of a problem in the fastest way
- enabling of computing larger problems
- more results in the same time
- efficient usage of shared computational resources like HERO and FLOW
energy efficiency
→ optimized better for environment
- cost reduction
- simple things can speedup your code significantly!
(e.g. change of compiler can give a speedup of ~2)

Motivation

However...

- the optimization of code can be time consuming
- for best performance may need deep knowledge of hardware architecture/programming
- readability of source code can be lower
- changes in source code needs again test phase

Strategies

Selection of the programming language

- script languages (python, R) are typically slow
→ subroutines (written in C, C++,...) could be fast
- C, Fortran give more performance
- C++ could simply lead to inefficient code if you not know what you are doing

Selection of compiler and compiler flags

- simplest way to increase the performance without changing the code
- compiler can give hints where are bottlenecks (e.g. missing SIMD parallelization)

Strategies

Code optimization

- profiling the program
- recoding of intensive used routines
- may replace own code by functions of highly optimized libraries like Intel Math Kernel Library (including BLAS, LAPACK, FFTW,...)
- change of algorithms
- parallelization

Good way for doing code optimization

- extract intensive functions to small prototypes
- after optimization transfer code back

OPTIMIZATION COMPILER & FLAGS

Compiler and flags

Coming back to profiling test program...

- measured wall clock times with different compilers and flags
- maybe there exist for different compiler better options for this problem

Compiler and flags

Compiler	Flags	wall clock [s]	speedup
gcc		6.02	1.00
gcc	-O3 -march=native -mtune=native	2.08	2.89
gcc	-Ofast -IPa -march=native -mtune=native	1.75	3.44
icc		1.13	5.33
icc	-O3 -xhost	1.11	5.42
icc	-O3 -xhost -ipo -align	0.785	7.67
icc	-fast -xhost -align	0.446	13.5
pgcc		1.80	3.34
pgcc	-O3 -fastsse	1.68	3.58
openc		1.59	3.79
openc	-Ofast -ipa -ffast-math	0.546	11.0
clang		4.59	1.31
clang	-O3	1.62	3.71
<i>python</i>		77.8	0.08

OPTIMIZATION

CODE OPTIMIZATION

Code optimization

Routine `initMatrix` (49% of wall clock time, $N=10000$)

```
/* init the matrix A with values */
void initMatrix(double *const A, const int N)
{
    int i,j;

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            A[i + j*N] = i/(j+1.);
}
```

Problems?

Code optimization

Routine `initMatrix` (49% of wall clock time, $N=10000$)

```
/* init the matrix A with values */
void initMatrix(double *const A, const int N)
{
    int i,j;

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            A[i + j*N] = i/(j+1.);
}
```

Problems

- jump through memory (counter i is outer loop) leads to cache misses (CPU loads *cache lines*, e.g. 64b blocks)

Code optimization

Routine `initMatrix` (49% of wall clock time, $N=10000$)

```
/* init the matrix A with values */  
void initMatrix(double *const A, const int N)  
{  
    int i,j;  
  
    for (j=0; j<N; j++)  
        for (i=0; i<N; i++)  
            A[i + j*N] = i/(j+1.);  
}
```

Problems

- jump through memory (counter i is outer loop) leads to cache misses (CPU loads *cache lines*, e.g. 64b blocks)
→ solution: swap loop for i and j

Code optimization

Routine `initMatrix` (49% of wall clock time, $N=10000$)

```
/* init the matrix A with values */
void initMatrix(double *const A, const int N)
{
    int i,j;

    for (j=0; j<N; j++)
        for (i=0; i<N; i++)
            A[i + j*N] = i/(j+1.);
}
```

Problems

- jump through memory (counter i is outer loop) leads to cache misses (CPU loads *cache lines*, e.g. 64b blocks)
→ solution: swap loop for i and j
- division in inner loop, no SIMD parallelization, need many cycles

Code optimization

Routine `initMatrix` (49% of wall clock time, N=10000)

```
/* init the matrix A with values */
void initMatrix(double *const A, const int N)
{
    int i,j;

    for (j=0; j<N; j++)
    {
        const double h = 1.0/(j+1.);
        for (i=0; i<N; i++)
            A[i + j*N] = i*h;
    }
}
```

- jump through memory (counter `i` is outer loop) leads to cache misses (CPU loads *cache lines*, e.g. 64b blocks)
→ solution: swap loop for `i` and `j`
- division in inner loop, no SIMD parallelization, need many cycles
→ solution: define compute division in outer loop

Code optimization

Influence of optimization on wall clock time

Compiler	Flags	wall clock [s]	speedup
gcc		2.76	1.00
gcc	-O3 -march=native -mtune=native	1.24	2.22
gcc	-Ofast -IPa -march=native -mtune=native	1.70	1.62
icc		1.19	2.32
icc	-O3 -xhost	1.30	2.12
icc	-O3 -xhost -ipo -align	0.666	4.14
icc	-fast -xhost -align	0.677	4.08
pgcc		1.08	2.56
pgcc	-O3 -fastsse	0.77	3.57
openc		0.783	3.52
openc	-Ofast -ipa -ffast-math	0.560	4.93
clang		2.30	1.20
clang	-O3	0.80	3.43

Code optimization

routine matVecMul (40% of wall clock time, N=10000)

```
/* y = alpha*A.x + beta*y */
void matVecMul(const double alpha, const double beta,
               const double *A, const double *x,
               double *const y, const int N)
{
    int i,j;

    for (i=0; i<N; i++)
        y[i] = beta*y[i];

    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
            y[i] = y[i] + alpha * A[i + j*N] * x[j];
}
```

Problems

- jump through memory (counter i is outer loop) leads to cache misses → solution: swap loop for i and j

Code optimization

Influence of optimization on wall clock time

Compiler	Flags	wall clock [s]	speedup
gcc		2.19	1.00
gcc	-O3 -march=native -mtune=native	1.23	1.78
gcc	-Ofast -IPa -march=native -mtune=native	0.518	4.23
icc		0.459	4.77
icc	-O3 -xhost	0.452	4.85
icc	-O3 -xhost -ipo -align	0.451	4.86
icc	-fast -xhost -align	0.444	4.93
pgcc		0.764	2.87
pgcc	-O3 -fastsse	0.499	4.39
openc		0.487	4.50
openc	-Ofast -ipa -ffast-math	0.546	4.01
clang		1.81	1.21
clang	-O3	0.466	4.70

Code optimization

- using of caches are important for performance
 - go linear through memory or use at least cache lines
 - avoid arbitrary jumps
- Intel compiler can give good hints by options, e.g. by `-vec-report` and/or `-opt-report`
- mathematical functions except `+`, `-`, `*` usually costly
- branches in inner loops are costly

OPTIMIZATION USAGE OF LIBRARIES

Usage of libraries

- there exist many libraries with optimized code for standard operations, e.g.
 - BLAS (vector/Vector, vector/matrix, matrix/matrix operations)
 - LAPACK (linear equation system solvers, eigenvalue solver)
 - FFTW (Fourier transformations)
 - highly optimized BLAS, LAPACK, FFTW
 - Intel MKL (Math Kernel Library)
 - ACML (AMD Core Math Library)
 - SparsePack
 - NAG (Mathematical library)
 - NetCDF/HDF5 (I/O)
 - PetSc (linear solvers)
 - GSL (GNU Scientific Library)
 - ...

Usage of libraries

- Example: replace `matVecMul` by BLAS

```
/* y = alpha*A.x + beta*y */  
void matVecMul(const double alpha, const double beta,  
              const double *A, const double *x,  
              double *const y, const int N)  
{  
    int inc = 1;  
    char Trans = 'N';  
  
    dgemv(&Trans, &N, &N, &alpha, A, &N, x, &inc, &beta, y, &inc);  
}
```

Compiler	Flags	wall clock [s]	speedup
gcc/BLAS	-Ofast -IPA -march=native -mtune=native	0.851	6.98
icc/MKL	-fast -xhost -align	0.523	11.4
pgcc/ACML	-fastsse	0.538	11.0

Summary

- debugging tools exist for various problems
 - tracking code
 - detecting errors like memory issues
- profiling enables performance measurements of functions/code fragments
 - give hints for code optimization
- optimization can significantly improve performance
- optimization can be done by
 - selection of compiler
 - compiler flags
 - code based optimization
 - change of algorithms
 - usage of libraries

Thanks a lot for your attention!

For further information please visit the HPC Wiki

<http://wiki.hpcuser.uni-oldenburg.de>

Exercises

- Login to FLOW/HERO

```
ssh -XY abcd1234@flow.hpc.uni-oldenburg.de
```

```
ssh -XY abcd1234@hero.hpc.uni-oldenburg.de
```

- Try to optimize sample code
- Change order of matrix $A[i + j*N] \rightarrow A[j + i*N]$ to measure performance