MASS, MASSV & ESSL 4.3

MASS and MASSV

- Three libraries provide elementary math functions:
  - C/Fortran intrinsics
  - MASS/MASSV (Math Acceleration Subroutine System)
  - ESSL/PESSL (Engineering Scientific Subroutine Library)
- Language intrinsics are the most convenient, but not the best performers

The Elementary functions included…

- MASS
  - **sqrt, rsqrt, exp, log, sin, cos, tan, atan, atan2, sinh, cosh, tanh, dnint, x**y**
- MASSV
  - **cos, dint, exp, log, sin, log, tan, div, rsqrt, sqrt, atan**

Comparison of standard lib and MASS intrinsic functions

Function	Sum from libm.a	Clock-cycles	Sum from libmass.a	Clock-cycles
sqrt	3.34427772158389e+11	159.0	3.34427772158389e+11	40.0
rsqrt	9.88776148452464e+01	189.0	9.88776148452464e+01	35.0
exp	2.22314235567424e+26	177.0	2.22314235567424e+26	65.0
log	1.10235345203187e+08	306.5	1.10235345203187e+08	95.0
sin	7.61032543425560e+04	217.6	7.61032543425560e+04	75.4
cos	1.81730644467472e+05	200.5	1.81730644467472e+05	73.4
tan	-6.62879483877644e+06	307.5	-6.62879483877644e+06	90.1
Atan	-2.53424519590047e+05	207.6	-2.53424519590047e+05	120.9
sinh	2.79285108669777e+24	273.4	2.79285108669777e+24	76.0
cosh	1.88661487104410e+26	244.6	1.88661487104410e+26	71.0
atan2	-7.56021669449783e+02	398.2	-7.56021669449782e+02	141.6
pow	3.72981324493266e+29	627.1	3.72981324493266e+29	171.0

Comparison of libm and MASSV functions

Libm function	Sum	Clock-cycles	MASSV function	Sum	Clock-cycles
div	2.35022308885783e+07	29.1	vdiv	2.35022308885783e+07	5.5
div	3.82109600477247e-03	29.0	vrec	3.82109600477247e-03	4.1
dsrt	3.30047180089010e+11	159.1	vsqrt	3.30047180089010e+11	11.2
rsqrt	9.83390477971166e+01	189.0	vrsqrt	9.83390477971166e+01	6.5
cos,sin	4.95000000000000e+06	429.6	vsincos	4.95000000000000e+06	57.7
Sin	-1.16545301554582e+05	217.9	vsin	-1.16545301554582e+05	32.2
Cos	-5.20893404460221e+04	203.7	vcos	-5.20893404460221e+04	32.1
Exp	3.31109589135987e+26	177.1	vexp	3.31109589135987e+26	18.9
log	1.08946996172333e+08	308.0	vlog	1.08946996172333e+08	20.7

Libm, MASS and MASSV

- No discernable difference in result –
  - **Exception:** atan2 difference in 14th significant place between libm & MASS

What are ESSL and Parallel ESSL?

- The Engineering and Scientific Subroutine Library (ESSL) family of products is a state-of-the-art collection of mathematical subroutines.
- Running on IBM Power servers and clusters, the ESSL family provides a wide range of high-performance mathematical functions for a variety of scientific and engineering applications

What Products are available?

- ESSL 4.3 contains over 500 high-performance serial and SMP mathematical subroutines tuned for Power4, Power4+, Power5, Power5+, Power6, PPC 970 and PowerPC 450 processors
- Parallel ESSL 3.3 contains over 125 high-performance SPMD mathematical subroutines specifically designed to exploit the full power of clusters of Power servers connected with a high performance switch

What Operating Systems are supported?

- ESSL 4.3
  - AIX 6.1
  - AIX 5.3
  - AIX 5.2
  - SLES10
  - RHEL5

What ESSL Libraries are Available?

- Thread-Safe Serial and SMP Libraries
  - 32 bit integers/32 bit pointers
  - 32 bit integers/64 bit pointers
  - 64 bit integers/64 bit pointers

What mathematical areas are supported?

- ESSL
  - Linear Algebra Subprograms
  - Matrix Operations
  - Linear Algebraic Equations
  - Eigensystems Analysis
  - Fourier Transforms, Convolution & Correlation & Related Computations
  - Sorting & Searching
  - Interpolation
  - Numerical Quadrature
  - Random Number Generation

What applications are supported?

- Callable from FORTRAN, C, and C++
- 32-bit integer, 32-bit pointer application support
- 32-bit integer, 64-bit pointer application support
- 64-bit integer, 64-bit pointer application support (ESSL Only)
- SMP Libraries are OpenMP based
- BLAS and Parallel BLAS Compatibility
- LAPACK and ScaLAPACK Compatibility

What do you get?

- ESSL
  - Libraries
  - Header File for C and C++
  - Manpages
  - Guide and Reference (Internet)
  - Install Guide (Internet)
  - Installation Verification Programs

How do you use ESSL?

- Create a source program or change an existing source program to call ESSL subroutines
- Compile the program
- Correct compiler-detected user errors
- Link-edit, load, and run the program
- Debug the program to isolate run-time errors
- Validate the program against test data
- Change the program and/or compiler options to improve performance
- Run the final version of the program to do work

What techniques are used to obtain high performance?

- SMP Algorithms
- SIMD Algorithms (e.g., VMX, BG/P PPC450D)
- Block Algorithms
  - Data Reuse (Data Caches and TLB)
- Data Prefetching
- Minimize Stride
  - If enough computations, copy to temporary space if used more than once
- Loop unrolling in computational kernels
  - Fully utilize the 2 Floating-Point Units, 2 Load-Store Units, and Floating-Point Registers
  - Careful scheduling of loops to avoid pipeline stalls

How usable are ESSL and Parallel ESSL?

- Easy to Use Call Interface
  - Fortran oriented but header file provided to assist C and C++ users
  - Dynamic allocation of work space
- Easy to obtain high performance
  - Replace key computational kernels with calls to math subroutines. As applications are run on new platforms simply relink to obtain high performance
  - Obtain high performance on SMP processors by relinking serial applications with ESSL SMP (Open MP) Library
- Informative and Flexible Error Handling
  - Messages are readily understandable - reference material not required
  - Single comprehensive message when all MPI tasks detect the same error
- Comprehensive Documentation
  - HTML, PDF and manpages available on the Internet
  - Quickly retrieve information
  - Organized according to the tasks performed
  - Readable by a wide class of users
- Easy to Install and Service

What about Migration?

- Long History of easy migrations
  - Customer applications almost always migrate to new releases and versions with no source code changes
  - Customer applications migrate to new hardware with no source code changes
- New XLF and VAC Compilers supported when they GA
- New AIX Operating System releases supported at GA (ESSL)

What's new in ESSL 4.3?

- POWER6
- Serial and SMP Libraries with 64 bit ints/64 bit ptrs
- VMX Support on Power6 and JS21
- 29 New LAPACK Subroutines
- RHEL5

What new subroutines are in ESSL 4.3?

- SGECON, DGECON, CGECON, ZGECON
  - Estimate the Reciprocal of the Condition Number of a General Matrix
- SPOCON, DPOCON, CPOCON, ZPOCON
- SPPCON, DPPCON, CPPCON, ZPPCON
  - Estimate the Reciprocal of the Condition Number of a Positive Definite Real Symmetric or Complex Hermitian Matrix
- SLANGE, DLANGE, CLANGE, ZLANGE
  - General Matrix Norm
- SLANSY, DLANSY, CLANHE, ZLANHE
- SLANSP, DLANSP, CLANHP, ZLANHP
  - Real Symmetric or Complex Hermitian Matrix Norm
- CPPTRI, ZPPTRI
  - Positive Definite Complex Hermitian Matrix Inverse
- SGEQRF, CGEQRF, ZGEQRF
  - General Matrix QR Factorization

Note on Core files

- Core files are text files. Look at the core file with a text editor, focus on the function call chain; feed the hex addresses to addr2line.
  - addr2line -e your.x hex_address
  - tail -n 10 core.511 | addr2line -e your.x
- Use grep and word-count (wc) to examine large numbers of core files:
  - grep hex_address “core.*” | wc -l

MPI_bug1

- Compile and execute mpi_bug1
- EXPLANATION: mpi_bug1 demonstrates how miscoding even a simple parameter like a message tag can lead to a hung program. Verify that the message sent from task 0 is not exactly what task 1 is expecting. Matching the send tag with the receive tag solves the problem.

MPI_bug2

- Compile and execute mpi_bug2
- EXPLANATION: mpi_bug2 shows another type of miscoding. The data type of the message sent by task 0 is not what task 1 expects. Nevertheless, the message is received, resulting in a segmentation fault or abnormal termination - depending upon the AIX version. Matching the send data type with the receive data type solves the problem.

MPI_bug3

- Compile and execute mpi_bug3
- EXPLANATION: mpi_bug3 shows what happens when the MPI environment is not initialized or terminated properly. Inserting the MPI init and finalize calls in the right locations will solve the problem.

MPI_bug4

- Compile and execute mpi_bug4
- Number of MPI tasks must be divisible by 4.
- EXPLANATION: mpi_bug4 shows what happens when a task does not participate in a collective communication call. In this case, task 0 needs to call MPI_Reduce as the other tasks do

MPI_bug5

- Compile and execute mpi_bug5
- EXPLANATION: mpi_bug5 demonstrates an unsafe program, because sometimes it will execute fine, and other times it will fail. The reason why the program fails or hangs is due to buffer exhaustion on the receiving task side, as a consequence of the way IBM has implemented an eager protocol for messages of a certain size. This subject is discussed in more detail in the MPI Performance Topics tutorial. One possible solution is to include an MPI_Barrier call in the both the send and receive loops.

MPI_bug6

- Compile and execute mpi_bug6
- Requires 4 MPI tasks.
- EXPLANATION: mpi_bug6 has a bug that will terminate the program under AIX, but be ignored under Intel Linux. The problem is that task 2 performs a blocking operation, but then hits the MPI_Wait call near the end of the program. Only the tasks that make non-blocking calls should hit the MPI_Wait. The coding error in this case is easy to fix - simply make sure task 2 does not encounter the MPI_Wait call.