Workshop on
High Performance Computing (HPC08)
School of Physics, IPM
February 16-21, 2008

HPC tools: an overview

Stefano Cozzini
CNR/INFM Democritos and
SISSA/eLab
cozzini@democritos.it

Agenda

- Tools for HPC
  - Debugging and Debuggers & compiler
  - Libraries
- How to use them efficiently:
  - A working example: the usage of optimized libraries on SMP platform (see exercise #7 and #12 on the wiki)

Compilers for Linux

- Free/Open Source:
    - GNU http://www.gnu.org/ (Fortran 77, C, C++, ...)
- Commercial:
    - PGI (Fortran 77, Fortran 90, C, C++) http://www.pgroup.com/
    - Intel (Fortran 77/95, C/C++) (individual Linux license free of charge)
    - PathScale (Fortran 77, Fortran 90, C, C++) http://www.pathscale.com (x86_64)
    - NAG http://www.nag.co.uk
    - Lahey http://www.lahey.com/
    - Absoft http://www.absoft.com/
- Almost all allow you a 15 day evaluation license

How to choose a compiler for HPC?

- Efficiency/ Parallelism
  - Does it produce efficient code? Does it produce correct code?
  - Is it able to exploit the hardware?
  - Does it support common language extensions or OpenMP directives?
- Availability/Cost
  - How much does it differ from the GNU compilers?
- Interoperability
  - Does it operate with other tools/compiler/languages?
- Utilities / Tools
  - Does it have a Debugger/ Profiler / other utilities?
- Diagnostic Capabilities
  - Is it able to detected errors/bugs in programs?
- Documentation/ support /training..

Debugging and debuggers..

- Identifying the cause of an error and correcting it
- Once you have identified defects, you need to:
  - find and understand the cause
  - remove the defect from your code
- Statistics show 60% of bug 'fixes' are not correct,

  ----> remove the symptom, but not the cause
- Improve productivity by getting it right the first time
- A lot of programmers don't know how to debug!
- Debugging needs practice and experience:

  ----> understand the science and the tools

Debugging (2)

- Debugging is a last resort:
  - Doesn't add functionality
  - Doesn't improve the science
- The best debugging is to avoid bugs:
  - Good program design
  - Follow good programming practices
  - Always consider maintainability and readability of code over getting results fast
  - Maximize modularity and code re-use

Errors are opportunities..

- Learn from the program you're working on:
  - Errors mean you didn't understand the program. If you knew it perfectly, it wouldn't have an error. You would have fixed it already
- Learn about the kind of mistakes you make:
  - If you wrote the program, you inserted the error
  - Once you find a mistake, ask youself:
    - Why did you make it?
    - How could you have found it more quickly?
    - How could you have prevented it? Are there other similar mistakes in the code?

---> Better to correct them now!

Debugging Tools

- **Source code comparator**
  - helps you find where you changed the code
  - look at diff, vimdiff, tkdiff, emacs/ediff program on UNIX
- **Compiler warning messages**

  Set the **compiler warning level** to the highest level, and fix the code so that it doesn't produce any warnings!

  Treat warnings as errors

  Compile and check warnings issued by a different compiler as well !
- **Execution Profiler**

  Programmer errors can cause bad performance as well as bad output

  Identify routines that take up a disproportionate amount of execution time
- **Debuggers: gdb, dbx, idb, pdbg, ddd (GUI)**

Why a debugger ?

- Better than print statements
- Allows to stop/start/single step execution
- Look at data and modify it
- 'Post mortem' analysis from core dumps
- Prove / disprove hypotheses
- Easier to use with modular code
- No substitute for good thinking
- But, sometimes good thinking is not a substitute for a good debugger!

Using a debugger

- When compiling use -g option to include debug info in object (.o) and executable
- 1:1 mapping of execution and source code only with disabled optimization

---> problem when optimization uncovers bug

- GNU compilers allow -g with optimization

--> not always correct line numbers

--> variables/code can be 'optimized away'

Gdb

- A GNU source level debugger
  - portable
  - efficient
  - it has some GUI (ddd)
- usage:

```
This is the GNU debugger.  Usage:
  gdb [options] [executable-file [core-file or process-id]]
Options:
 --batch              Exit after processing options.
 --cd=DIR             Change current directory to DIR.
 --command=FILE       Execute GDB commands from FILE.
 --core=COREFILE      Analyze the core dump COREFILE.
 --directory=DIR      Search for source files in DIR.
 --exec=EXECFILE      Use EXECFILE as the executable.
 --fullname           Output information used by emacs-GDB interface.
 --mapped             Use mapped symbol files if supported on this system.
 --nw                 Do not use a window interface.
 --nx                 Do not read .gdbinit file.
 --readnow            Fully read symbol files on first access.
 --se=FILE            Use FILE as symbol file and executable file.
 --symbols=SYMFILE    Read symbols from SYMFILE.
 --version            Print version information and then exit.
For more information, type "help" from within GDB, or consult the GDB manual (available as
    on-line info or a printed manual).Report bugs to "bug-gdb@prep.ai.mit.edu".gdbh]
```

Some Standard Mathematical libraries

- Vendor’s library
  - ESSL Engineering and scientific subroutine library (IBM)
  - MKL (Intel)
  - ACML (AMD)
- ISV
  - IMSL International Mathematical and Statistical Libraries
  - NAG   Numerical Algorithm group (UK labs)
- Free
  - **NETLIB**   a WWW metalibrary of free math software
  - **SLATEC** comprehensive Mathematical and statistical Package
  - **LAPACK/BLAS** Linear algebra package
  - **CERN** European center for nuclear research
  - **Petsc:** ODE/PDE parallel solvers
  - **FFTW:** fft library

Why use libraries ?

- Efficiency:
  - Better to use routines written by professionals. Math libraries are designed to use the “tricks of coding” to use the CPU in the most efficient way not necessarily the most straight forward..
- Parallelism for free:
  - On modern multicore SMP machine vendors provide multi threaded version of same libraries.
- Portability:
  - Use of standardized libraries improves portability to other platforms

LAPACK: Linear Algebra Package

- 1992: Dongarra et al.
- Supersedes and extends Eispack and Linpack packages...
  - Eispack Linpack: well-tuned for old machine
- It has been designed to be efficient on a wide range of modern high performance computer:
  - vector processing
  - risc workstations
  - shared memory multiprocessors
- It uses very efficient KERNELS : BLAS library

BLAS: Basic Linear Algebra Subprograms

- IDEA: create a standard to identify the basic set of operations involved in linear algebra problems;
- Objectives:
  - Accuracy;
  - Efficiency;
  - Portability;
  - Maintainability;
- Dates: 70’s; ===> vector programming
  Fortran programming

Basic Linear Algebra Subroutines

Name	Description	Examples
Level-1 BLAS	Vector Operations	$C = \sum X_i Y_i$
Level-2 BLAS	Matrix-Vector Operations	$B_i = \sum_k A_{ik} X_k$
Level-3 BLAS	Matrix-Matrix Operations	$C_{ij} = \sum_k A_{ik} B_{kj}$

Efficiency: q parameter

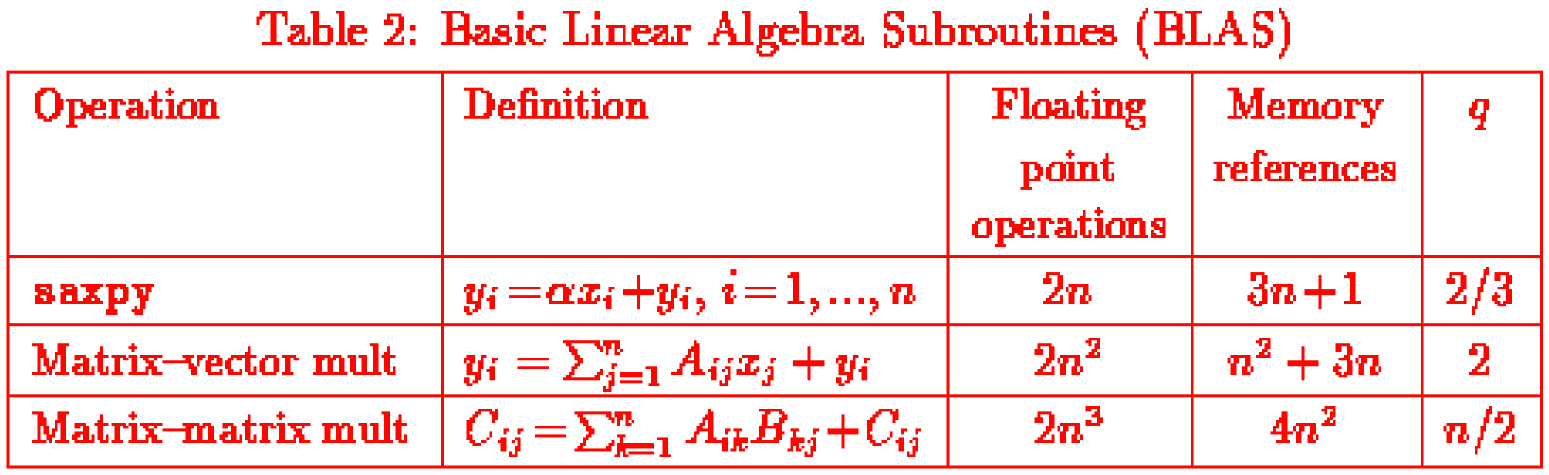


Table 2: Basic Linear Algebra Subroutines (BLAS)

Operation	Definition	Floating point operations	Memory references	q
saxpy	$y_i = \alpha x_i + y_i,\ i = 1, \ldots, n$	$2n$	$3n+1$	$2/3$
Matrix–vector mult	$y_i = \sum_{j=1}^{n} A_{ij} x_j + y_i$	$2n^2$	$n^2 + 3n$	2
Matrix–matrix mult	$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj} + C_{ij}$	$2n^3$	$4n^2$	$n/2$

The parameter q is the ratio of flops to memory references. Generally:

1. Larger values of q maximize useful work to time spent moving data.
2. The higher the level of the BLAS, the larger q.

It follows...

- BLAS1 are memory bounded ! (for each computation a memory transfer is required)
- BLAS2 are not so memory bounded (can have good performance on super-scalar architecture)
- BLAS3 can be very efficient on super-scalar computers because not memory bounded

OPTIMIZATION TRICK:

Write your vector-matrix operations as matrix-matrix operation if possibile

Optimized libraries

- Standard BLAS and LAPACK (not machine specific)
  - Still better then naïve code..
- Intel Math Kernel Library (MKL): INTEL
  - BLAS, LAPACK,FFT and many other (scaLAPACK...)
  - Current version 10.0.x
- AMD Math Core Library (ACML): AMD
  - BLAS LAPACK FFT and many others
  - Current version 4.0
- Automatically Tuned Linear Algebra Software (ATLAS):
  - BLAS and some LAPACK routines that can be compiled on PC based machines to obtain better maximum performance by tuning machine specific parameters.
  - Current version 3.8.0 (http://math-atlas.sourceforge.net/)

scaLAPACK

- 1995: Dongarra et. al.version 1.0 of
- Scalable Linear Algebra PACK AGE
  - (now version 1.7):
- parallel MP-implementation of LAPACK:

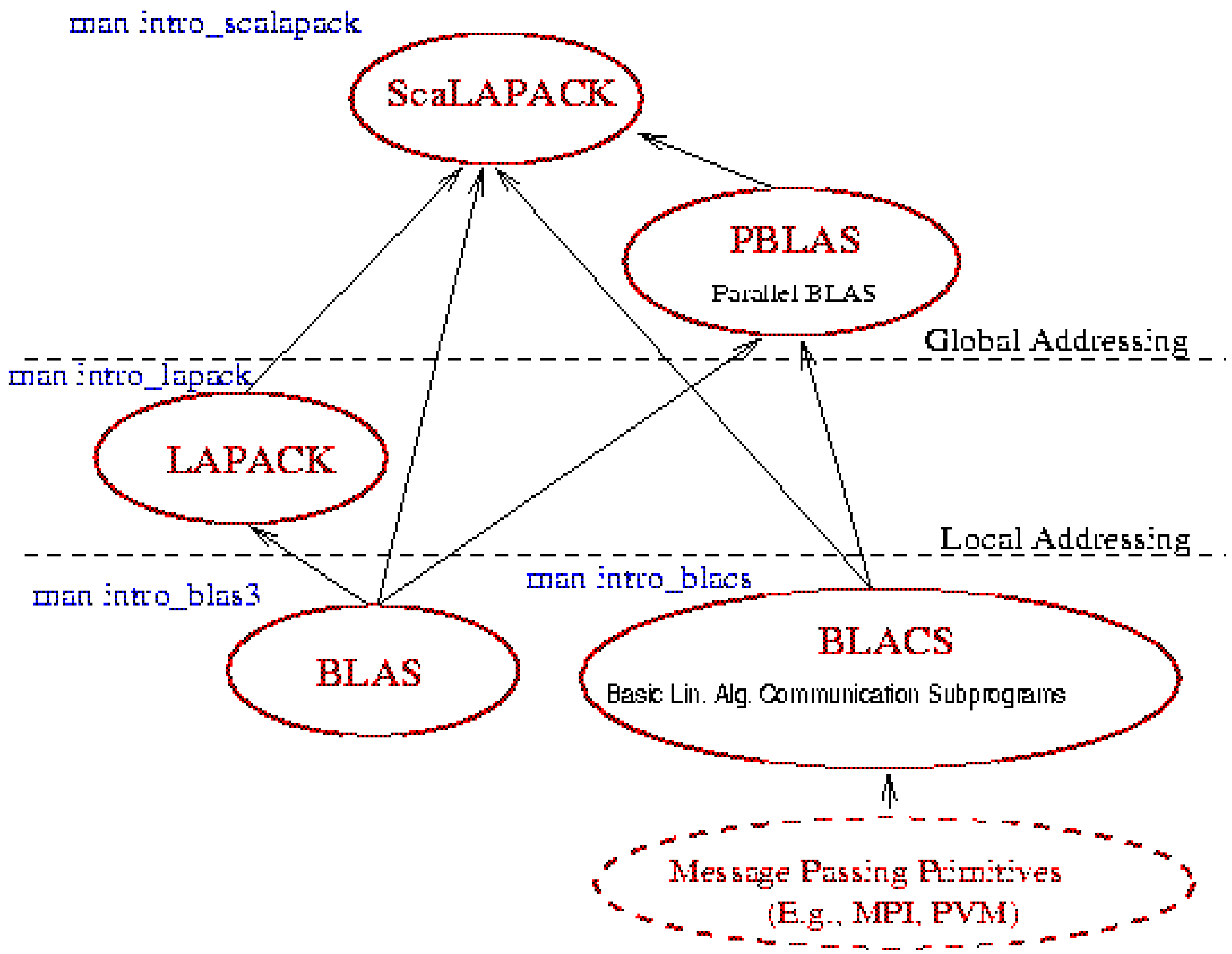
- From FAQ:

The **ScaLAPACK** (or Scalable LAPACK) library includes a subset of **LAPACK** routines redesigned for distributed memory MIMD parallel computers. It is currently written in a Single-Program-Multiple-Data style using explicit message passing for interprocessor communication.

LAPACK and ScaLAPACK

	LAPACK	ScaLAPACK
Machines	Workstations, Vector, SMP	Distributed Memory, DSM
Based on	BLAS	BLAS, BLACS
Functionality	Linear Systems Least Squares Eigenproblems	Linear Systems Least Squares Eigenproblems (less than LAPACK)
Matrix types	Dense, band	Dense, band, out-of-core
Error Bounds	Complete	A few
Languages	F77 or C	F77 and C
Interfaces to	C++, F90	HPF
Manual?	Yes	Yes
Where?	www.netlib.org/ lapack	www.netlib.org/ scalapack

scaLAPACK: components



How to use libraries...

- In this exercise I propose to test three different implementations of the DGEMM routine blas3
- A small F77 program driver to the DGEMM routine is provided together with a Makefile.
- You will link this driver against three different libraries: the original BLAS one, the ATLAS library and the Intel MKL library.

Matmul: a driver for DGEMM

```
      tnow = second()
*this is the original dgemm
      call dgemm('No','No',n,n,n,1.0d0,B, \\
             ldm,C,ldm,1.0d0,A,ldm)
      time(nr) = time(nr) + second() - tnow
```

Makefile for matmul

```
#simple makefile for the matmul program
CC = gcc
CFLAGS = -O3

FC = g77   <----- gfortran
FFLAGS  = -O3

# please indicate Blas libraries for matmul1

LIB1 = -L/home/cozzini -latlas_pgi

# please indicate Blas libraries for matmul2

LIB2 = -L/usr/local/lib -lgoto_p4_512p-r0.9
-lpthread
LIB2 = -L/opt/intel/mkl/9.0/lib/32 -lmkl_ia32
-lguide -lpthread
default: matmul0 matmul1 matmul2
```

conclusions

- Use libraries whenever you can:
  - Enhanced performance
  - Great portability