FORTRAN and MPI

Message Passing Interface (MPI)

Day 5

DANISH CENTER FOR APPLIED MATHEMATICS AND MECHANICS

Course plan:

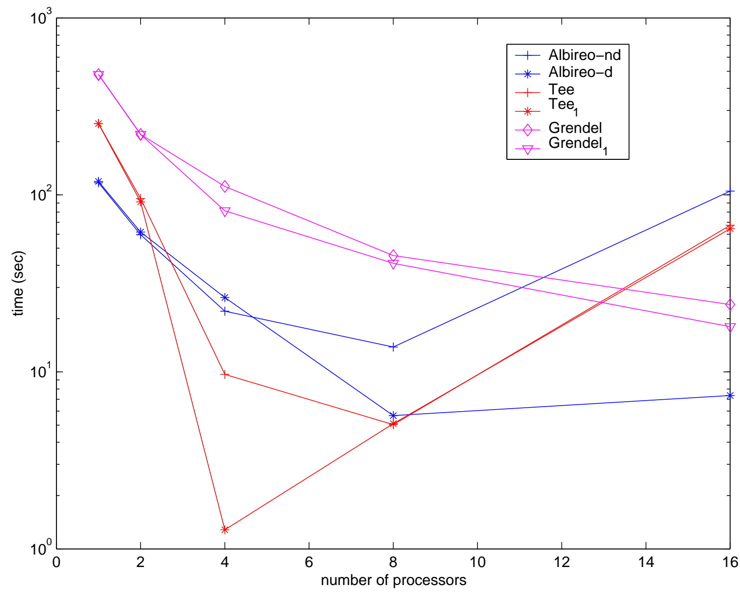
- MPI - General concepts
- Communications in MPI
  – Point-to-point communications
  – Collective communications
- Parallel debugging
- Advanced MPI: user-defined data types, functions
  – Linear Algebra operations
- Advanced MPI: communicators, virtual topologies
  – Parallel sort algorithms
- Parallel performance. Summary. Tendencies

Computational and communication complexity

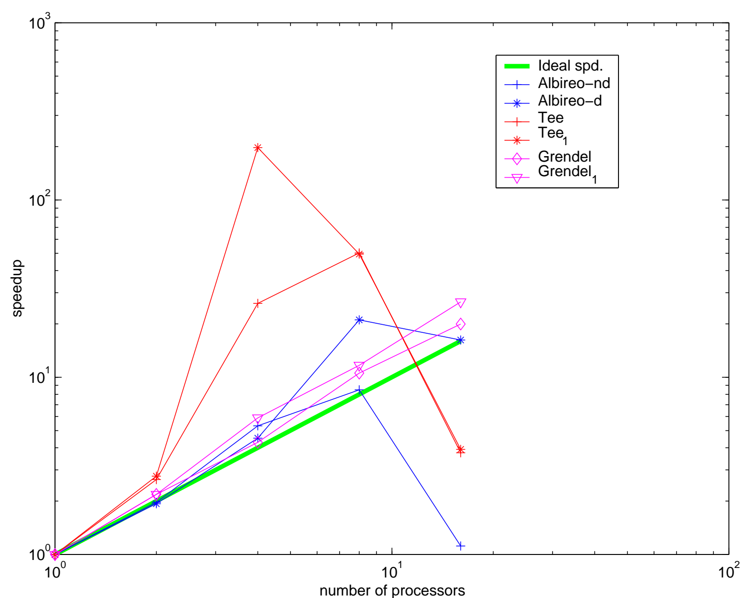
- notations, notions
- parallel performance metrics
- parallel performance models
- computational complexity of some basic algorithms/operations

Basic terminology

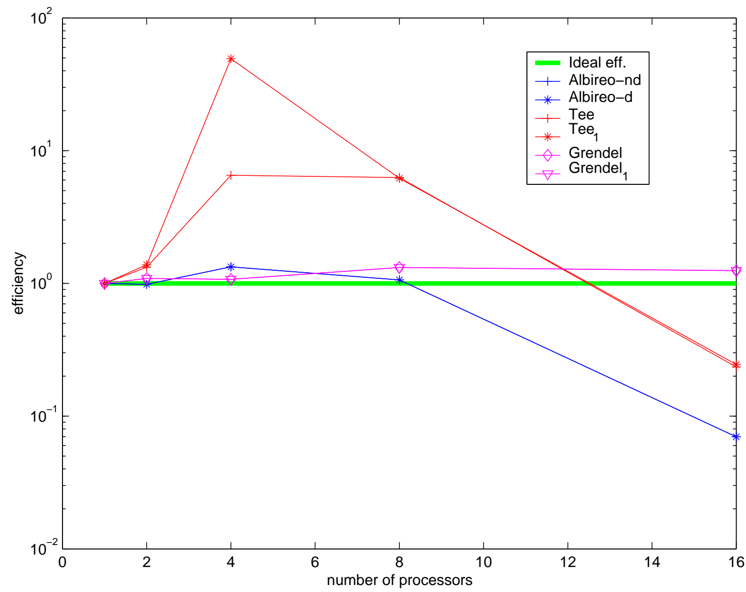
- parallel machine (homogeneous), number of PE p, size of the problem N, some algorithm A
- computational complexity $W(A,p)$, $W(A,1)$
- clock cycle
- execution time
  serial: $T(A,1) = t_c W(A)$
  parallel: $T(A,p) = T_s(A) + \frac{T_p(A)}{p} + T_c(A,p)$
- FLOPS rate (peak performance: *theoretical* vs *sustained*)



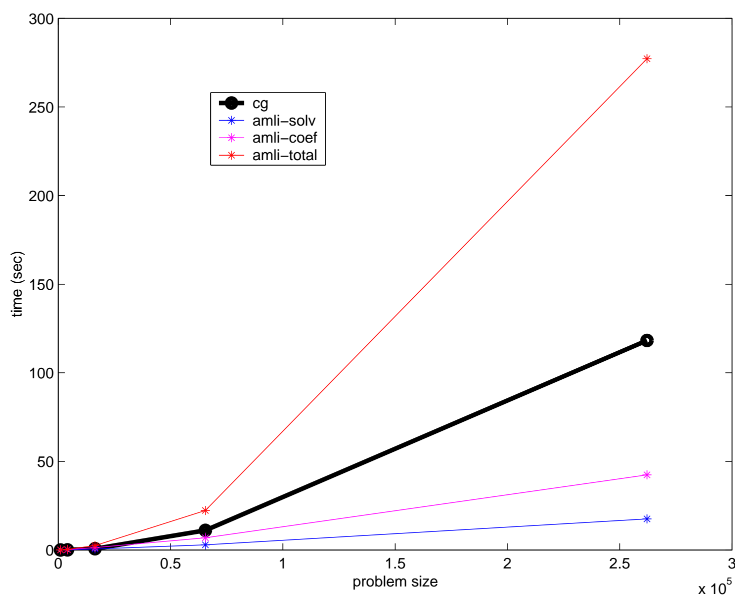
Three parallel computers: execution times



Three parallel computers: speedup curves



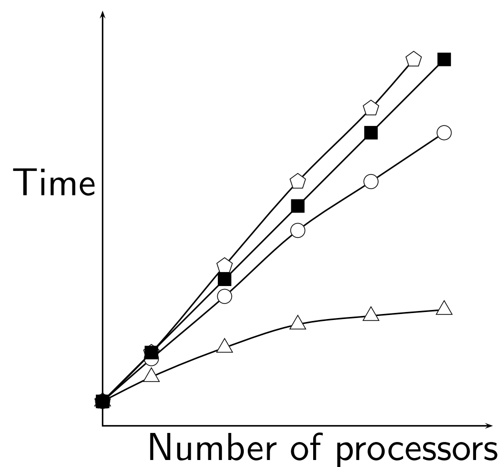
Three parallel computers: parallel efficiency



Two algorithms on Albireo

Parallel performance metrics

- $T(A,p)$ is the primary metric !!!
- speedup $S(A,p) = \frac{T(A,1)}{T(A,p)} \leq p$; relative, absolute
- efficiency $E(A,p) = \frac{S(A,p)}{p} \leq 1$
- redundancy $W(A,p)/W(A,1)$
- scalability



Examples of speedup curves

Measuring *speedups* - pros and cons: *contra*- relative speedup is that it "hides" the possibility for $T(A,1)$ to be very large. The relative speedup "favors slow processors and poorly-coded programs" because of the following observation.

Let the execution times on a uni- and p-processor machine, and the corresponding speedup be $T_0(A,1)$ and $T_0(A,p)$ and $S_0 = \dfrac{T_0(A,1)}{T_0(A,p)} > 1$.
Next, consider the same algorithm and optimize its program implementation. Then usually
$T(A,p) < T_0(A,p)$ but also $S < S_0$.
Thus, the straightforward conclusion is that worse programs have better speedup.

A closer look: $T(A,p) = \beta T_0(A,p)$ for some $\beta < 1$. However, $T(A,1)$ is also improved, say $T(A,1) = \alpha T_0(A,1)$ for some $\alpha < 1$.
What might very well happen is that $\alpha < \beta$. Then, of course, $\dfrac{S_0}{S} = \dfrac{\beta}{\alpha} > 1$.

When the comparison is done via the absolute speedup formula, namely

$$\frac{\widetilde{S}_0}{\widetilde{S}} = \frac{T(A^*,1)}{T_0(A,p)}\frac{T(A,p)}{T(A^*,1)} = \beta < 1.$$

In this case $T(A^*,1)$ need not even be known explicitly. Thus, the absolute speedup does provide a reliable measure of the parallel performance.

Both speedup and efficiency, as well as MFLOPSrate, are tools for analysis but not a goal of parallel computing.
None of these alone is a sufficient criterion to judge whether the performance of a parallel system is satisfactory or not. Furthermore, there is a tradeoff between the parallel execution time and the efficient utilization of many processors, or between efficiency and speedup. One way to observe this is to fix N and vary p. Then for some p_1 and p_2 we have the relation

$$\frac{E(A,p_1)}{E(A,p_2)} = \frac{p_2 T(A,p_2)}{p_1 T(A,p_1)}.$$

If we want $E(A,p_1) < E(A,p_2)$ and $T(A,p_1) > T(A,p_2)$ to hold simultaneously, then $\frac{p_2}{p_1} < \frac{T(A,p_1)}{T(A,p_2)}$, i.e., the possibility of utilizing more processors is limited by the gain in execution time. As a realistic goal, when developing parallel algorithms for massively parallel computer architectures one aims at efficiency which tends to one with both increasing problem size and number of processors.

Scalability

* ***scalability of a parallel machine***: The machine is scalable if it can be incrementally expanded and the interconnecting network can incorporate more and more processors without degrading the communication speed.

* ***scalability of an algorithm***: If, generally speaking, it can use all the processors of a scalable multicomputer effectively, minimizing idleness due to load imbalance and communication overhead.

* ***scalability of a machine-algorithm pair***

How to define scalability?

Definition 1: *A parallel system is* scalable *if the performance is linearly proportional to the number of processors used.*
BUTS: impossible to achieve in practice

Definition 2: *A parallel system is scalable if the efficiency $E(A,p)$ can become bigger than some given efficiency $E_0 \in (0,1)$ by increasing the size of the problem, i.e., $E(A,p)$ stays bounded away from zero when N increases (efficiency-conserving model).*

Definition 3: *A parallel system is scalable if the parallel execution time remains constant when the number of processors p increases linearly with the size of the problem N (time-bounded model).* **BUTS:** too much to ask for since there is communication overhead.

Definition 4: *A parallel system is scalable if the achieved average speed of the algorithm on the given machine remains constant when increasing the number of processors, provided that the problem size is increased properly with the system size.*

Presuming an algorithm is parallelizable, i.e., a significant part of it can be done concurrently, we can achieve large speed-up of the computational task using

(a) well-suited architecture;
(b) well-suited algorithms;
(c) well-suited data structures.

A degraded efficiency of a parallel algorithm can be due to either the computer architecture or the algorithm itself:

(i) lack of a perfect degree of parallelism in the algorithm;
(ii) idleness of computers due to synchronization and load imbalance;
(iii) of the parallel algorithm;
(iv) communication delays.

More on parallel performance

Definition: A parallel system is said to be *cost-optimal* if the cost of solving a problem in parallel is proportional to the execution time of the fastest-known sequential algorithm on a single processor.

The cost is understood as the product pT_p

Example: Adding n numbers on a p-processor machine ($p < n$).
The serial complexity of adding n numbers is $O(n)$. On a p-processor hypercube ($p = 2^d$) the complexity becomes

$$O(\frac{n}{p} + 2\log p).$$

n	p=1	p=4	p=8	p=16	p=32
64	1.0	0.80	0.57	0.33	0.17
192	1.0	0.92	0.80	0.60	0.38
320	1.0	0.95	0.87	0.71	0.50
512	1.0	0.97	0.91	0.80	0.62

Scaled speedup:
Compare scalability figures when problem size **and** number of PEs are increased simultaneously in a way that the load per individual PE is kept large enough and approximately constant.

Parallel performance models

- The fundamental principle of computer performance; Amdahl's law (1967)

  Given: N operations, grouped into k subtasks $N_1, N_2, \cdots, N_k$, which must be done sequentially, each with rate R_i.

$$T = \sum_{i=1}^{k} t_i = \sum_{i=1}^{k} \frac{N_i}{R_i} = \sum_{i=1}^{k} \frac{f_i\, N}{R_i}; \quad \overline{R} = \frac{T}{N} N / \sum (f_i N / R_i) = \frac{1}{\sum_{i=1}^{k} f_i / R_i}$$

  Hence, the average rate $\overline{R}$ for the whole task is the weighted harmonic mean of $R_1, R_2, \ldots, R_k$.

  For the special case of only two subtasks - f_p (parallel) and $1 - f_p$ - serial, then

$$\overline{R}(f_p) = \frac{1}{\frac{f_p}{R_p} + \frac{1-f_p}{R_s}} \quad \text{and} \quad S = \frac{p}{f_p + (1-f_p)p} \leq \frac{1}{1-f_p}.$$

Gene Amdahl:

> *For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution...The nature of this overhead (in parallelism) appears to be sequential so that it is unlikely to be amendable to parallel processing techniques. Overhead alone would then place an upper limit on throughput on five to seven times the sequential processing rate, even if the housekeeping were done in a separate processor...At any point in time it is difficult to foresee how the previous bottlenecks in a sequential computer will be effectively overcome.*

- Gustafson-Barsis law (1988):

  Perhaps, the first breakthrough of the Amdahl's model is the result achieved by the 1988 Gordon Bell's prize winners - a group from Sandia Laboratories.

  On a 1024 processor nCUBE/10 and with f_p computed to be in the range of $(0.992, 0.996)$ they encountered a speedup of 1000 while the Amdahl's law prediction was only of the order of 200 ($S = 1024/(0.996 + 0.004 * 1024) \approx 201$).

$$\begin{aligned} T(A,1) &= (1 - f_p) + f_p p \\ T(A,p) &= (1 - f_p) + f_p = 1 \quad \text{properly scaled problem} \\ S &= T(A,1) = p - (p-1)(1 - f_p) \end{aligned}$$

Grid size	Coarsest level No (total no. of levels)	Number of PEs							Time (sec)
		1	2	4	8	16	32	64	
64^2	8(12)	60.14	31.32	18.19					total
		59.47	31.03	18.09					outer
		25.68	13.51	8.30					coars.
		0.11	1.17	1.26					comm.
128^2	10(14)		161.76	79.77	41.96				total
			160.76	79.09	41.66				outer
			89.18	45.61	24.00				coars.
			2.46	2.67	3.29				comm.

The Stokes problem: Performance results on the Cray T3E-600 computer

Grid size	Coarsest level No (total no. of levels)	Number of PEs							Time (sec)
		1	2	4	8	16	32	64	
256^2	8(16)			455.67					total
				452.31					outer
				101.34					coars.
				6.26					comm.
	10(16)			406.62	190.54	94.61	49.55	28.90	total
				403.49	189.06	93.86	49.18	28.71	outer
				159.75	80.09	41.63	21.97	13.20	coars.
				5.31	5.93	5.58	4.62	3.89	comm.
	12(16)			440.90	213.34	107.87	56.79	34.21	total
				438.01	211.96	107.16	56.43	34.04	outer
				283.38	142.39	74.42	39.19	24.25	coars.
				5.75	7.32	7.41	5.90	4.49	comm.
	14(16)			824.96					total
				822.33					outer
				728.07					coars.
				15.03					comm.

The Stokes problem: Performance results on the Cray T3E-600 computer (cont)

Grid size	Coarsest level No (total no. of levels)	Number of PEs							Time (sec)
		1	2	4	8	16	32	64	
512^2	12(18)					632.60 629.44 363.38 14.28	304.24 302.71 183.18 12.14	154.65 153.81 96.15 10.14	total outer coars. comm.
1024^2	12(20)						1662.73 1655.73 810.11 29.89	829.71 826.22 422.25 22.26	total outer coars. comm.
	14(20)						1913.08 1906.57 1326.37 33.19		total outer coars. comm.

The Stokes problem: Performance results on the Cray T3E-600 computer (cont)

MPI is not the only alternative...

- PVM - Parallel Virtual machine (1989)
- MPI - Message Passing Interface (1993)
- BSP - Bulk Synchronous Parallel computations (1990)

PVM

The major difference:
- MPI is a library to write an application program, not a distributed memory operating system;
- PVM can be regarded as the 'de facto' standard for distributed computing.

The general notion for PVM is that of a *virtual machine*, which is a set of heterogeneous hosts, connected with some network, which appears logically to the user as a single large parallel computer.

The problem is decomposed into separate programs, each compiled to run on a specific type of computer. The major issue becomes how to exchange data between the programs. The communication is done by calls to PVM library routines (`pvm_send()` and `pvm_recv()` through message buffers. One data exchange may contain data items of various types and they are packed and unpacked respectively upon `send` and `receive`.

host:	a physical machine (or nodes of a parallel machine)
virtual machine:	meta-machine composed of one or more hosts
task:	a PVM process - smallest unit of computation
TID:	a unique identifier associated with each task
PVMD:	the PVM daemon, e.g. 'pvm3/lib/ARCH/pvmd3' where ARCH is the architecture of the host such as sol2, sun, sgi.
message:	ordered list of data between tasks
group:	set of tasks assigned a symbolic name, each task has a unique index in the group and a task can belong to zero or more groups

Want to know more? Visit `http://www.csm.ornl.gov/pvm/`. Maintained at The Computer Science and Mathematics Division (CSM), Oak Ridge National Lab.

BSP

The claim: *BSP* is a 'bridging' model between software and hardware for parallel computation that is analogous to the von Neumann model of sequential computation.

The *bulk synchronous parallel (BSP)* model consists of:

- a set of processor-memory pairs,
- a communication network that delivers messages *point-to-point*,
- a mechanism for efficient synchronization of all, or a subset of, the processors.

... superstep

The parameters that define performance:

- s - processor speed (number of steps per second, or the actual rate at which useful calculation is done);
- p - number of processors;
- ℓ - the time the machine needs for the barrier synchronization (depends on network latency);
- g - the cost, in steps per word, of delivering message data, calculated from the average cost of transferring each word of messages of all sizes in the presence of other traffic on the network (bandwidth inefficiency, gap)

The closer the g value approaches 1 and the smaller the ℓ value is for a given system, the easier it is to produce scalable parallel performance on that architecture.

Advantages of the BSP model

* a higher level of abstraction for the programmer
a cost model for performance analysis and prediction
which is simpler and compositional,
more efficient implementations on many machines
!! Predictability

BSPonMPI

http://www.bsp-worldwide.org/
http://www.bsp-worldwide.org/implmnts/oxtool/
http://bsponmpi.sourceforge.net/bench/index.html

Scalar product: MPI vs BPS

```
 ------------------------------------------------------------
c Scalar product:          MPI                          --> BSP
c ------------------------------------------------------------
      double precision function dot1(x,y,n)
      implicit none
      include 'mpif.h'                         --> include 'fbsp.h'
      integer n,i,ierr,comm                    --> integer n,i,field_length
      double precision lsum,gsum,x(n),y(n)
      lsum=0.0d0                               --> field_length = 8
      do i=1,n
         lsum=lsum + x(i)*y(i)
      enddo
      call MPI_ALLREDUCE(lsum,gsum,1,MPI_DOUBLE_PRECISION,
     >                  MPI_SUM, MPI_COMM_WORLD, ierr)
c-->   call BSP_REDUCE(SUM,lsum,gsum,field_length)
      mpi_dot1 = gsum
      return
      end
```

OpenMP

OpenMP is an industry-wide standard for directive-based parallel programming on SMP (Symmetric MultiProcessor) systems.

OpenMP Is an *Application Program Interface (API)* enabling explicit direct multi-threaded, shared memory parallelism.

OpenMP is of three primary API components:

- compiler directives;
- runtime library routines;
- environment variables.

OpenMP is portable: the API is specified for C/C++ and Fortran.

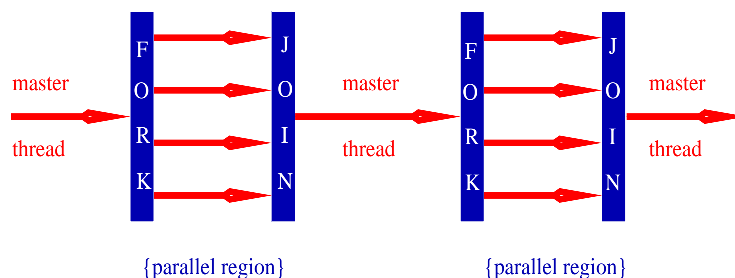
OpenMP is standardized: jointly defined and endorsed by a group of major computer hardware and software vendors. Expected to become an ANSI standard later.

OpenMP is Not:

- meant for distributed memory parallel systems (by itself);
- necessarily implemented identically by all vendors;
- guaranteed to make the most efficient use of shared memory (currently there are no data locality constructs).

http://www.openmp.org

Thread Based Parallelism:	A shared memory process can consist of multiple threads. OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm.
Explicit Parallelism:	OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
Fork - Join Model:	OpenMP uses the fork-join model of parallel execution:



C / C++ - General Code Structure

```
#include <omp.h>
      main ()  {
      int var1, var2, var3;
      Serial code
            ...
            ...
      Beginning of parallel section. Fork a team of threads.
      Specify variable scoping

      #pragma omp parallel private(var1, var2) shared(var3)
        {
        Parallel section executed by all threads
              ...
              ...
        All threads join master thread and disband
        }
      Resume serial code
            ...
      }
```


Fortran - General Code Structure

```
      PROGRAM HELLO
      INTEGER VAR1, VAR2, VAR3
      Serial code
            ...
            ...
      Beginning of parallel section. Fork a team of threads.
      Specify variable scoping

!$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)
      Parallel section executed by all threads
            ...
            ...
      All threads join master thread and disband
!$OMP END PARALLEL

      Resume serial code
            ...
            ...
      END
```

```
      program sum

c Compute global sum

      call omp_set_num_threads(16)
      xsum = 0.0
c$omp parallel do private(i,m,x) shared(xsum)
      do i = 1, 800
        m = (i-1)/100
        x = m + 1.0
        xsum = xsum + x
      enddo
      print *, 'Global Sum = ', xsum

      end
```


MPI Java Wrapper:

Design:

◇ all set of classes with lightweight functional interface to MPI

◇ the classes are based on the fundamental MPI object types (communicators, groups, etc.)

◇ the Java wrapper language bindings provide a semantically correct interface to MPI

◇ one-to-one mapping between MPI and Java wrapper bindings

◇ to the greatest extent possible, the Java wrapper for MPI functions are methods functions of MPI classes

Structure:

◇ everything is included in the MPI package (Java term)

◇ A set of basic MPI functions has been implemented

init(), finalize(), Wtime(), Wtick(), Send(), Bsend(), Rsend(), Bcast(), Gather(), Scatter(), Allgather(), Reduce(),...

`http://www.jppf.org/index.php`
About JPPF (JAVA Parallel Processing Framework):
a grid toolkit for Java that makes it easy to run your applications in parallel, and speed up their execution by orders of magnitude.

Linux and Unix platforms (32 and 64 bits)
Windows 2000, XP, 2003 (32 and 64 bits)

Computational Grids enable

- sharing,
- selection,
- aggregation

of a wide variety of geographically distributed computational resources (supercomputers, compute clusters, storage systems, data sources, instruments, people)
and presents them as a single, unified resource for solving large-scale compute and data intensive computing applications (e.g, molecular modelling for drug design, brain activity analysis, and high energy physics).

PETSc – `http://acts.nersc.gov/petsc/`

PETSc, the **Portable, Extensible Toolkit for Scientific computation** provides sets of tools for the parallel (as well as serial), numerical solution of PDEs that require solving large-scale, sparse nonlinear systems of equations. PETSc includes nonlinear and linear equation solvers that employ a variety of Newton techniques and Krylov subspace methods. PETSc provides several parallel sparse matrix formats, including compressed row, block compressed row, and block diagonal storage.

PETSc is designed to facilitate extensibility. Thus, users can incorporate customized solvers and data structures when using the package. PETSc also provides an interface to several external software packages including BlockSolve95, ESSL, Matlab, ParMeTis, PVODE, and SPAI. PETSc is fully usable from Fortran, C and C++, and runs on most UNIX based-systems.

HYPRE High Performance Preconditioners

https://computation.llnl.gov/casc/linear_solvers/sls_hypre.html
Beta-version: hypre-2.2.0b from September 2007

HYPRE is a software library for solving large, sparse linear systems of equations on massively parallel computers. The library was created with the primary goal of providing users with advanced parallel preconditioners. Issues of robustness, ease of use, flexibility, and interoperability also play an important role.

TRILINOS - http://trilinos.sandia.gov/

The Trilinos Project is an effort to develop algorithms and enabling technologies within an object-oriented software framework for the solution of large-scale, complex multi-physics engineering and scientific problems.
A unique design feature of Trilinos is its focus on packages.

MUMPS: a MUltifrontal Massively Parallel sparse direct Solver

`http://graal.ens-lyon.fr/MUMPS/`

Distributed Multifrontal Solver (F90, MPI based);

Dynamic Distributed Scheduling to accommodate both numerical fill-in and multi-user environment;

Use of BLAS, ScaLAPACK. Version for complex arithmetic;

Parallel factorization and solve phases (uniprocessor version also available);

Iterative refinement and backward error analysis;

Various matrix input formats

High Performance Computing (HPC)

Where are we?

◇ Performance:
- Sustained performance has increased substantially during the last years.
- On many applications, the *price-performance* ratio for the parallel systems has become smaller than that of specialized supercomputers. But · · ·
- Still, some applications remain hard to parallelize well (adaptive methods).

◇ Languages and compilers:
- Standardized, portable, high-level languages exist (MPI, PVM, HPF, Matlab). But · · ·
  Initial releases of HPF were not very efficient (1993 - 1997).
- Message passing programming is tedious and hard to debug.
- Programming difficulties remain still a major obstacle for mainstream scientists to parallelize existing codes.

However, we are witnessing and we are given the chance to participate in the exiting process of parallel computing achieving its full potential power and solving the most challenging problems in Scientific Computing.