Introduction to supercomputing:

parallel efficiency of linear algebra algorithms for shared and distributed memory computers

Igor KONSHIN[1,2]

[1]Marchuk Institute of Numerical Mathematics of RAS
[2]Sechenov University

September 22–25, 2020

Plan

Situation:

- parallel computations are faster than serial ones...
- a lot of // results, but there is no // efficiency estimates
- it is sufficient // models, estimates exist as well,
  but too abstract or too complicated

Aim:

- constructive (workable) // efficiency estimates
  for algorithms (linear algebra)

Plan:

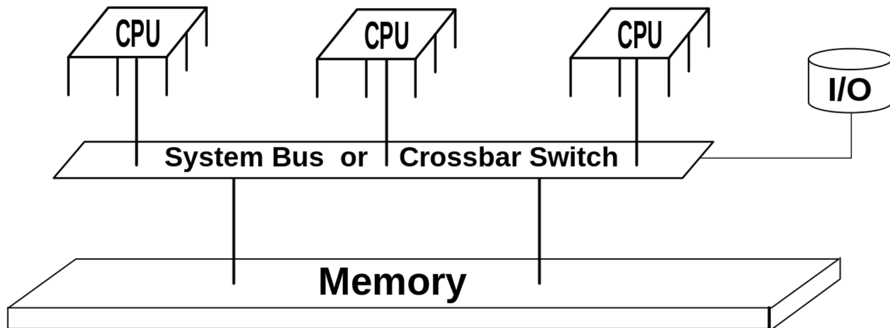
- computers classification
- shared and distributed memory
- parallel efficiency estimation for algorithms
- example of programs
- results of numerical experiments

Computer classification

- Shared memory (personal computers, notebooks, phones, ...)
- Distributed memory (clusters, supercomputers, ...)

¡ Let's start with 'Shared memory' !

Shared (or common) memory



OpenMP — shared memory

OpenMP (Open Multi-Processing) for C, C++, and Fortran

OpenMP consists of a set of

- macros,
- preprocessor directives,
- library functions, and
- environment variables.

1997: v.1.0
2018: v.5.0

// efficiency (common memory) [Amdahl's law, 1967]

- p – number of processes (threads)
- $T(p)$ – algorithm run time for p processes
- σ – the portion of sequential (not parallelized) operations

Speedup:

$$S(p) = \frac{T(1)}{T(p)} = \frac{T(1)}{\sigma T(1) + \frac{(1-\sigma)T(1)}{p}} = \frac{p}{1+\sigma(p-1)}$$

¡ *Example:* $S(p=0) = 1$, $S(\sigma=0) = p$, $S(\sigma=0.01, p\to\infty) = 100$ *!*

// (parallel) Efficiency:

$$E(p) = \frac{S(p)}{p} = \frac{1}{1+\sigma(p-1)}$$

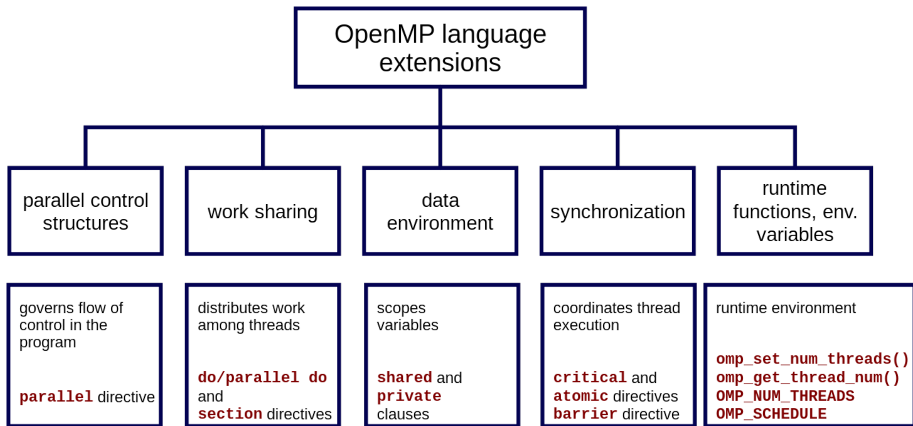
¡ *Directly applicable to OpenMP programs !*

[A'67] Amdahl G.M.: Validity of the single-processor approach to achieving large scale computing capabilities. In: AFIPS Conference Proceedings, vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483–485.
`http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf`

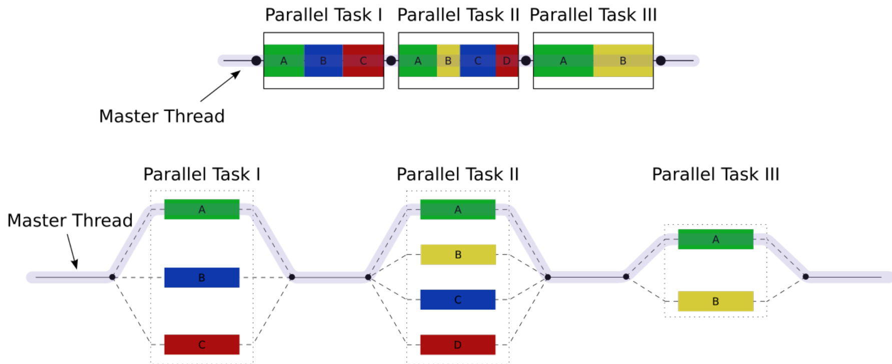
Conditions for the best applicability of Amdahl's law

- arithmetic operations are quite **uniform**;
- **all** available threads are involved in computing the parallel part of the code;
- **balance** of calculations for all threads;
- **scalability** of thread operation, i.e. the thread performance does not depend on the number of threads (or, in other words, the execution time of the parallel part does indeed decrease by p times when using p threads).

OpenMP: language extension



OpenMP: threads



OpenMP: instead of **#pragma omp parallel for**

```
#ifdef _OPENMP
#pragma omp parallel for
#endif
    for (i=0; i<n; i++) {...}

    FOR (i=0; i<n; i++) {...}

#ifdef _OPENMP
    #include <omp.h>
    #ifdef _WIN32 // MS Windows
        #define OMP_PARALLEL_FOR __pragma(omp parallel for)
    #else // Linux
        #define OMP_PARALLEL_FOR _Pragma("omp parallel for")
    #endif
#else // OpenMP is not supported
    #define OMP_PARALLEL_FOR
#endif

#define FOR(iii) OMP_PARALLEL_FOR for(iii)
```

Examples

- daxpy: $\alpha \cdot \vec{x} + \vec{y}$
- norm: $\|\vec{x}\|$
- mvm: $y = A \cdot x$

Source codes and // results

Shared and distributed memory

¡ Everyone can run these examples on // computer! !

INM RAS cluster

INM RAS cluster segments:

- x6core
- x8core
- x10core
- x12core

Segment **x6core:**

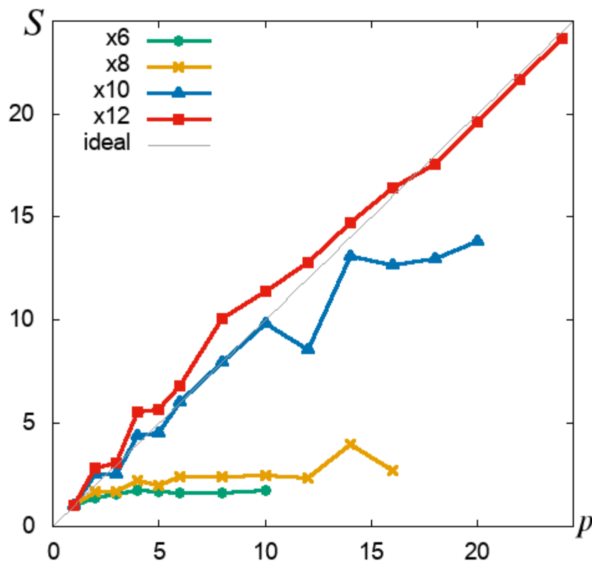
- Compute Node Asus RS704D-E6;
- 12 cores (two 6-cores processors Intel Xeon X5650@2.67GHz);
- RAM memory: 24 GB;
- Disc memory: 280 GB;
- Operating system: SUSE Linux Enterprise Server 11 SP1 (x86_64);
- Network: Mellanox Infiniband QDR 4x.

To build the code, we used the Intel C compiler version 4.0.1, with Intel MPI Library version 5.0.3.

OMP daxpy: $\alpha \cdot \vec{x} + \vec{y}$

```
#include <omp.h>

int main()
{
   t = omp_get_wtime();
#pragma omp parallel for
   for (i=0; i<N; i++) y[i] += a * x[i];
   t = omp_get_wtime() - t;

   printf("OMP daxpy: N=%d max_th=%d time=%lf\n",
          N, omp_get_max_threads(), t);
}

$ gcc -fopenmp omp_daxpy.c
$ export OMP_NUM_THREADS=4 ; a.out
OMP daxpy: N=1000000 max_th=4 time=0.01
```

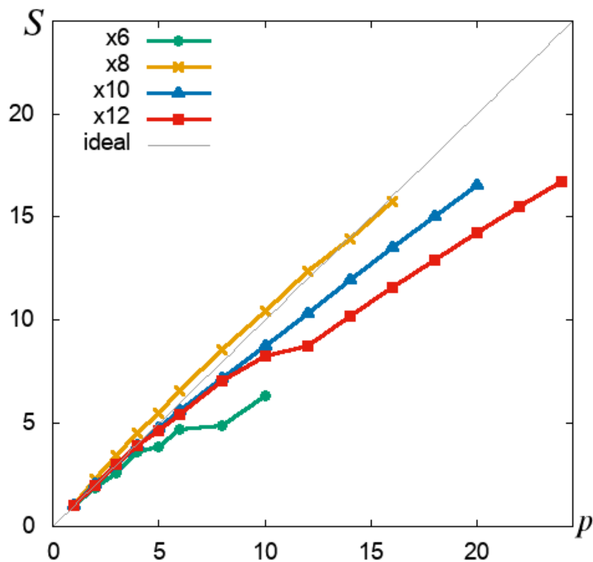
OMP daxpy: $\alpha \cdot \vec{x} + \vec{y}$



OMP norm: $\|\vec{x}\|$

```
#include <omp.h>

#define N 1000000

int main()
{
   t = omp_get_wtime();
   sum = 0.0;
#pragma omp parallel for reduction (+:sum)
   for (i=0; i<N; i++)
      sum += x[i] * x[i];
   sum = sqrt(sum);
   t = omp_get_wtime() - t;

   printf("OMP norm: N=%d max_th=%d time=%lf\n",
          N, omp_get_max_threads(), t);
}

$ gcc -fopenmp omp_norm.c -O -lm
$ export OMP_NUM_THREADS=4 ; ./a.out
OMP norm: N=1000000 max_th=4 time=0.01
```

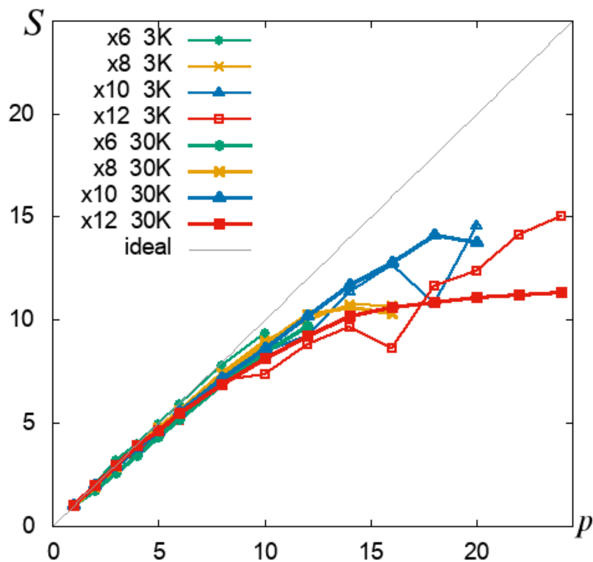
OMP norm: $\|\vec{x}\|$



OMP mvm: $y = A \cdot x$

```
#include <omp.h>

#define N 1000

int main()
{
   double a[N][N], x[N], y[N], t;

   t = omp_get_wtime();
#pragma omp parallel for private (j)
   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += a[i][j] * x[j];
   t = omp_get_wtime() - t;

   printf("OMP mvm: N=%d max_th=%d time=%lf\n",
          N, omp_get_max_threads(), t);
}

$ gcc -fopenmp omp_mvm.c -O
$ export OMP_NUM_THREADS=4 ; ./a.out
OMP mvm: N=4000 max_th=4 time=0.1
```

OMP mvm: $y = A \cdot x$



OMP paradigm

Suppose that several builders (computational threads) are working (performing arithmetic operations) in some not too large room. Due to the hustle (memory access conflicts) in a limited space (cash memory) or the limited speed of the construction resources delivery (RAM speed), delays can occur, up to a slowdown in program execution.

OpenMP: conclusions

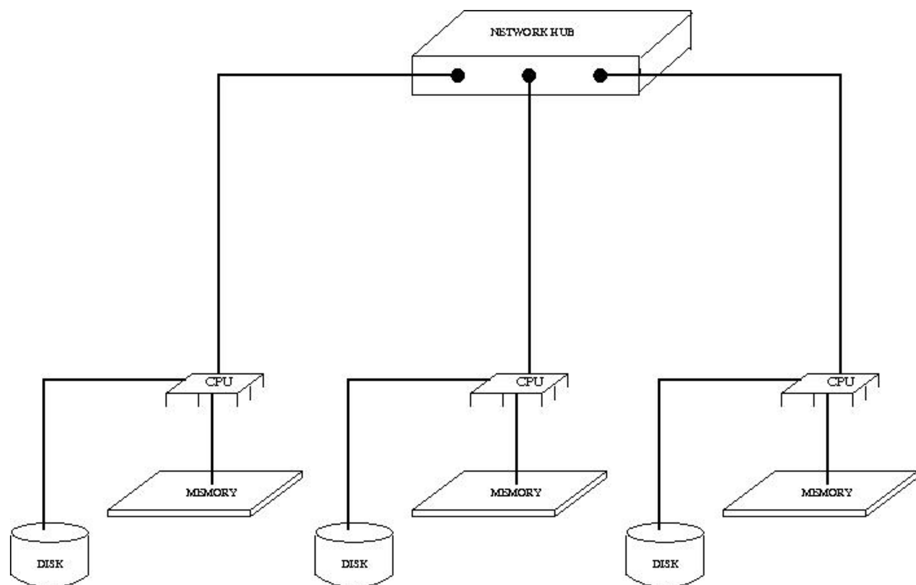
OpenMP — parallelization of *arithmetic*

Main features:

- shared memory
- compiler directives
- easy enough parallelization

Distributed memory

Distributed memory



MPI paradigm

Several submarines (MPI processes) float in an ocean, each one contains a sailor (for performing arithmetic calculations) and a radio operator (for sending/receiving messages).
Local data (local memory) are nearby, and nothing is known about data on other submarines (only submarine numbers are available).
One can work with your own data quite successfully (arithmetic is fast), but requesting (by Morse code) and then receiving data from neighboring submarines is a very long process (communications are relatively slow).
In this case, the main delays in work may occur due to the transmission of messages, and they should be taken into account when estimating the parallel efficiency of collaboration.

Distributed memory (MPI exchanges)

$$T_c = \tau_0 + \tau_c L_c$$

τ_0 – message initialization time
τ_c – message transfer rate (i.e., transfer time for a unit length message)
T_c – message transmission time for length L_c
Set for simplicity $\tau_0 = 0$, then

$$T_c = \tau_c L_c$$

Similarly,

$$T_a = \tau_a L_a$$

τ_a – execution time of one (characteristic) arithmetic operation
L_a – total number of algorithm arithmetic operations

// efficiency (distributed memory) [IK, 2012]

Let

$$\tau = \tau_c/\tau_a, \qquad L = L_c/L_a$$

are the characteristics of the 'parallelism' of the computer used and the algorithm under investigation, respectively

Then

$$\begin{aligned} S &= S(p) = T(1)/T(p) = T_a/(T_a/p + T_c/p) = pT_a/(T_a + T_c) \\ &= p/(1 + T_c/T_a) = p/(1 + (\tau_c L_c)/(\tau_a L_a)) = p/(1 + \tau L) \end{aligned}$$

and even easier for the // (parallel) efficiency:

$$E = \frac{S}{p} = \frac{1}{1 + \tau L}$$

Assumptions

- unlike the Amdahl's law formula, here it is considered that **all** calculations are parallelized and the fully sequential part of the algorithm is absent ($\sigma = 0$);
- delays in calculations occur **only** due to exchanges, and algorithms with synchronous exchanges are more suitable for this model;
- processors are loaded **uniformly** (or, in other words, the calculations are balanced);
- computing nodes are **homogeneous**;
- the rate of arithmetic calculations τ_a does not depend on p;
- the message transfer rate τ_c is also **independent** of p (this less obvious fact means the scalability of the communication network of the computer used).

MPI mvm: $y = A \cdot x$ (Speedup estimation)

$$Y_i = \sum_{j=1}^{n} A_{ij} X_j, \quad i = 1, ..., n$$

```
[:]    [== == ==]    [:]
---    ----------    ---
[:] =  [== == ==]  * [:]
---    ----------    ---
[:]    [== == ==]    [:]
```

¡ At each process: send local portion of x and collect the global x !

Let n be a matrix dimension, then:

$$L_a = n^2/p, \quad L_c = (n/p)(p-1), \quad L = L_c/L_a = (p-1)/n$$

$$S = p/(1 + (p-1)\tau/n).$$

If $n = 1000$ and $\tau = 10$, then

$$S(p=1) = 1, \qquad S(p=10) \approx 9, \qquad S(p=100) \approx 50.$$

MPI mvm: $y = A^T \cdot x$ (transposed matrix)

$$Y_i = \sum_{j=1}^{n} A_{ij}^T X_j, \quad i = 1, ..., n$$

```
[:]    [::|::|::]    [:]
---    [  |  |  ]    ---
[:] =  [::|::|::]  * [:]
---    [  |  |  ]    ---
[:]    [::|::|::]    [:]
```

¡ At each process: send local partial sums of y and collect the local x !

Let n be a matrix dimension, then:

$$L_a = n^2/p, \quad L_c = (n/p)(p-1), \quad L = L_c/L_a = (p-1)/n$$

$$S = p/(1 + (p-1)\tau/n).$$

¡ Algorithms are different, but estimations are identical! !

MPI mvm: $y = A \cdot x$ (band matrix)

```
[:]   [===     ]   [:]
---   ---------    ---
[:] = [  ===   ] * [:]
---   ---------    ---
[:]   [     ===]   [:]
```

¡ At each process: send local overlapping elements of x to neighbours !

Let n be a matrix dimension, r is the bandwidth of the matrix, then:

$$L_a = (2r+1)n/p, \quad L_c = 2r(p-1)/p, \quad L = (2r/(2r+1))(p-1)/n \approx (p-1)/n$$

$$S \approx p/(1 + (p-1)\tau/n).$$

¡ The same efficiency as for dense matrix! !

¡ Efficiency is independent of bandwidth r! !

MPI mvm: $y = A \cdot x$ (sparse multi-diagonal matrix)

```
[:]   [\\ \      ]   [:]
---   -----------   ---
[:] = [ \ \\\ \ ] * [:]
---   -----------   ---
[:]   [     \ \\]   [:]
```

¡ At each process: send local overlapping elements of x to neighbours !

Let n be a matrix dimension, r be the bandwidth, d is the number of diagonals, then:

$$L_a = dn/p, \quad L_c = 2r(p-1)/p, \quad L = 2r(p-1)/(dn)$$

$$S = p/(1 + 2r(p-1)\tau/(dn)).$$

¡ The efficiency as much less wrt. that for a band matrix! !

MPI — distributed memory

MPI (Message Passing Interface) for C, C++, and Fortran

MPI consists of a set of

- library routines, and
- environment variables.

1994: v.1.0
2015: v.3.1

MPI: concept

- Communicators:
  - MPI_COMM_WORLD
- Point-to-point communications:
  - MPI_Send
  - MPI_Recv
  - MPI_Isend, MPI_Irecv
- Collective communications:
  - MPI_Bcast
  - MPI_Gather
  - MPI_Reduce
  - MPI_Allreduce
  - MPI_Barrier
- Special functions:
  - MPI_Init
  - MPI_Finalize
  - MPI_Wtime

MPI_Send

```
int MPI_Send(
  void* buf,
    // the start address of the send message buffer
  int count,
    // the number of transmitted elements
  MPI_Datatype datatype,
    // type of transmitted elements
  int dest,
    // destination process number
  int msgtag,
    // message id
  MPI_Comm comm
    // group id
);
```

MPI_Recv

```
int MPI_Recv(
  void* buf,
    // (OUT) the start address of the receive message buffer
  int count,
    // the maximal number of received elements
  MPI_Datatype datatype,
    // type of transmitted elements
  int source,
    // sender process number
  int msgtag,
    // message id
  MPI_Comm comm,
    // group id
  MPI_Status *status
    // (OUT) received message parameters
);
```

MPI basic functions

```
#include <mpi.h>

int main (int argc, char **argv)
{
   int id, np;
   double t;

   MPI_Init (&argc, &argv);                   /* starts MPI */
   MPI_Comm_rank (MPI_COMM_WORLD, &id); /* get current process id */
   MPI_Comm_size (MPI_COMM_WORLD, &np); /* get total number of processes */

   MPI_Barrier(MPI_COMM_WORLD);              /* synchronize processes */
   t = MPI_Wtime();
   /* ... some code ... */
   MPI_Barrier(MPI_COMM_WORLD);
   t = MPI_Wtime() - t;                      /* get wall-clock time */

   MPI_Finalize();                           /* finalize MPI */
   return 0;
}

$ mpicc mpi_base.c
$ mpirun -np 4 ./a.out
```

MPI — versions

- 1994: MPI 1.0 (Send+Recv, asynchronous, pairwise bidirectional, collective exchanges)
- 1995: MPI 1.1
- 1997: MPI 2.0 (Put+Get = one-way communications, spawning MPI processes and threads, extended collective operations across multiple communicators)
- 2012: MPI 3.0 (MPI_Iallreduce, MPI_Ibarrier, ...)
- 2015: MPI 3.1

MPI Forum — `http://www.mpi-forum.org/`

MPI — implementations

- MPICH — https://ru.wikipedia.org/wiki/MPICH
- OpenMPI — https://www.open-mpi.org/
- Intel MPI — http://software.intel.com/en-us/intel-mpi-library/
- MPJ Express — http://www.mpj-express.org/ MPI Java

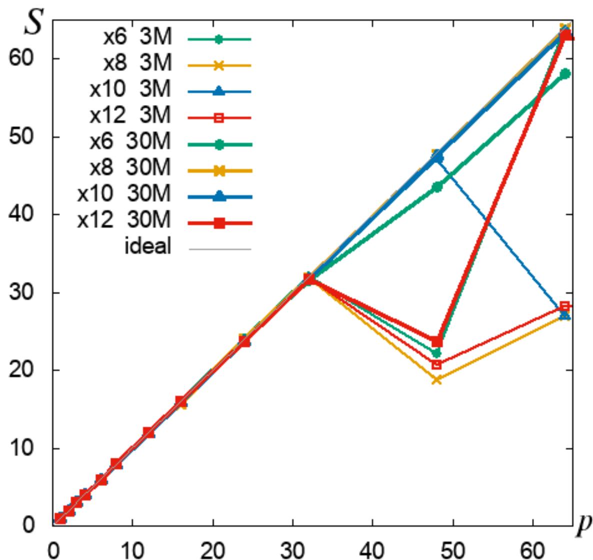
OpenMPI 3.1 currently contains over 400 features:

- MPI_Comm* (37)
- MPI_File* (61)
- MPI_Group* (14)
- MPI_Win* (39)
- MPIX_* (23) from future MPI 4.0

MPI daxpy: $\alpha \cdot \vec{x} + \vec{y}$

```
#define N 1000000 /* global vector dimension */

   double a, x[N/np], y[N/np], t, perf;

   MPI_Barrier(MPI_COMM_WORLD);
   t = MPI_Wtime();
# pragma omp parallel for  //<-- OpenMP
   for (i=0; i<N/np; i++)
      y[i] += a * x[i];
   MPI_Barrier(MPI_COMM_WORLD);
   t = MPI_Wtime() - t;
   perf = N / t * 1e-6;
   if (id == 0)
      printf("MPI daxpy: N=%d np=%d time=%lf perf=%lf MFLOPS\n",
             N, np, t, perf);


$ mpicc -O mpi_daxpy.c
$ mpirun -np 10 ./a.out
N=1000000 np=10 time=0.01 perf=100.0 MFLOPS
```

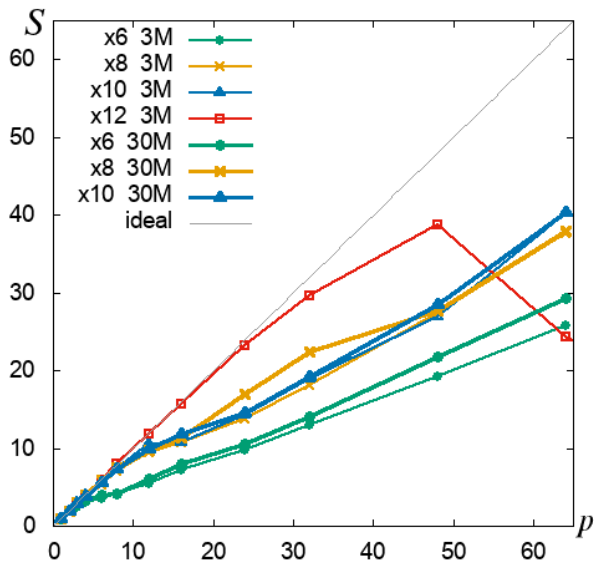
MPI daxpy: $\alpha \cdot \vec{x} + \vec{y}$



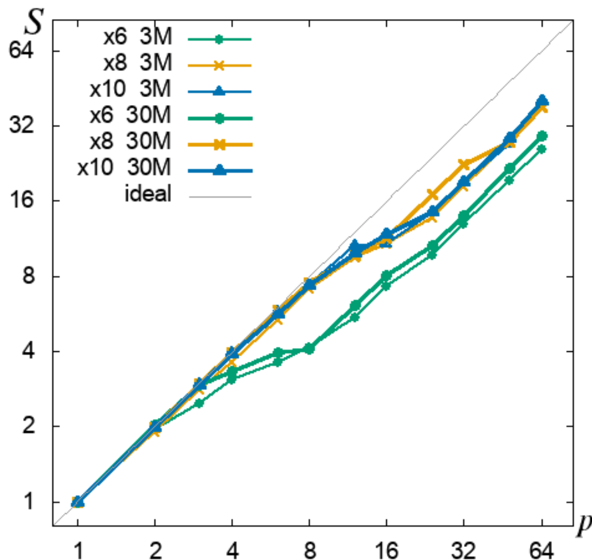
MPI norm: $\|\vec{x}\|$

```
#define N 1000000 /* global vector dimension */

   double sum, tmp, x[N/np], t, perf;

   MPI_Barrier(MPI_COMM_WORLD);
   t = MPI_Wtime();
   sum = 0.0;
   for (i=0; i<N/np; i++)
      sum += x[i] * x[i];
   tmp = sum;
   MPI_Allreduce(&tmp, &sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
   sum = sqrt(sum);
   MPI_Barrier(MPI_COMM_WORLD);
   t = MPI_Wtime() - t;
   perf = 1e-6 * N / t;
   if (id == 0)
      printf("MPI norm: N=%d np=%d norm=%f time=%lf perf=%lf MFLOPS\n",
             N, np, sum, t, perf);


$ mpicc -O mpi_norm.c
$ mpirun -np 10 ./a.out
N=1000000 np=10 norm=100.0 time=0.01 perf=100.0 MFLOPS
```

MPI norm: $\|\vec{x}\|$



MPI norm: $\|\vec{x}\|$ in log-scale

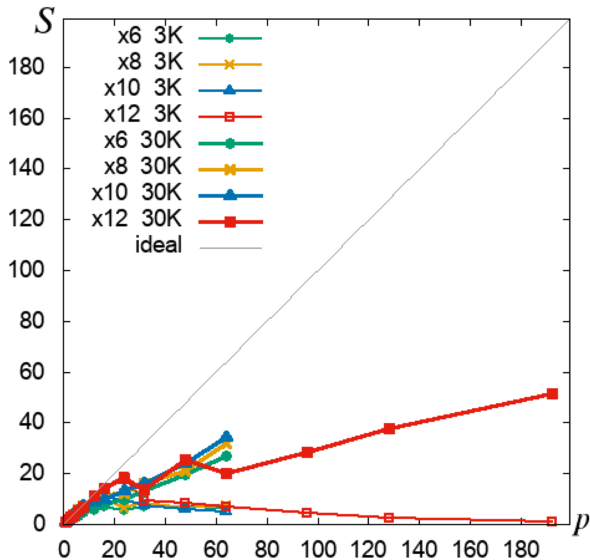


MPI mvm: $y = A \cdot x$

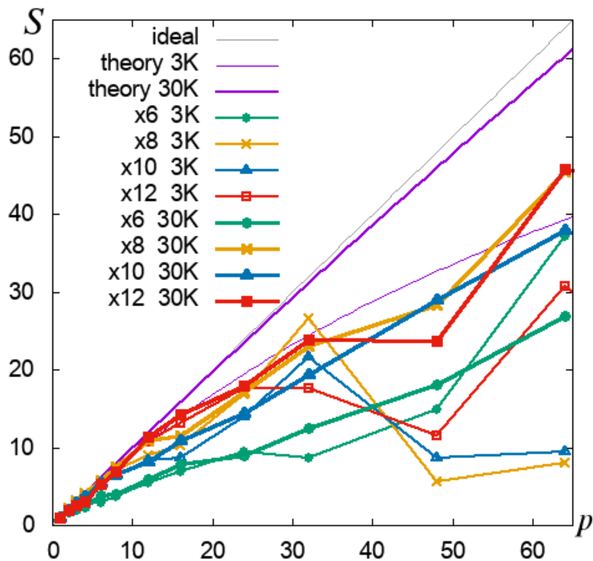
```
#define N 1000

   int i, j, n, id, np;
   double *a, *x, *y, *X, *ai, t;

   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &np);

   n = N / np;
   a = (double *) malloc(n*N * sizeof(double));
   x = (double *) malloc(n   * sizeof(double));
   y = (double *) malloc(n   * sizeof(double));
   X = (double *) malloc(N   * sizeof(double));

   MPI_Barrier(MPI_COMM_WORLD);
   t = MPI_Wtime();
   for (i=0; i<n; i++)
      X[i+id*n] = x[i];
   for (i=0; i<np; i++)
      MPI_Bcast(X+id*n, n, MPI_DOUBLE, i, MPI_COMM_WORLD);
   for (i=0; i<n; i++) {
      ai = a + i * N;
      for (j=0; j<N; j++)
         y[i] += ai[j] * X[j];
   }
   MPI_Barrier(MPI_COMM_WORLD);
   t = MPI_Wtime() - t;
   if (id == 0)
      printf("MPI mvm: N=%d np=%d time=%lf\n", N, np, t);
```

```
$ mpicc -O mpi_mvm.c
$ mpirun -np 4 ./a.out
MPI mvm: N=1000 np=4 time=0.1
```

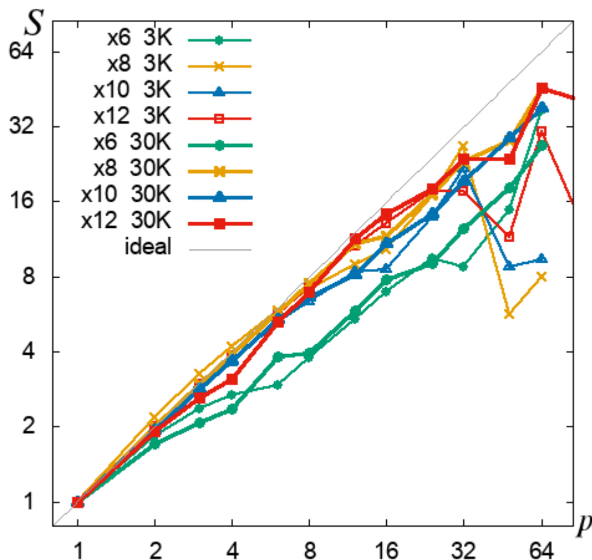
MPI mvm: $y = A \cdot x$



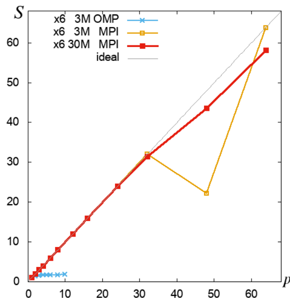
MPI mvm: $y = A \cdot x$

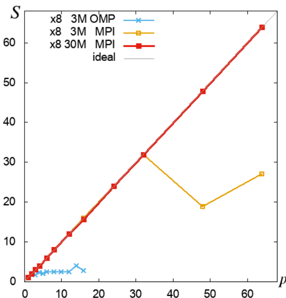


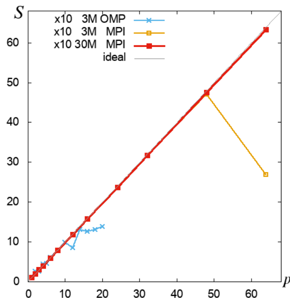
MPI mvm: $y = A \cdot x$

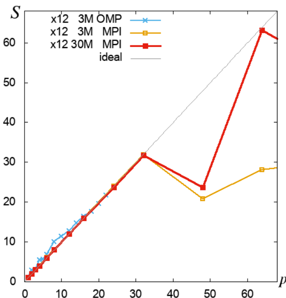


Comparison: OMP vs. MPI daxpy: $\alpha \cdot \vec{x} + \vec{y}$

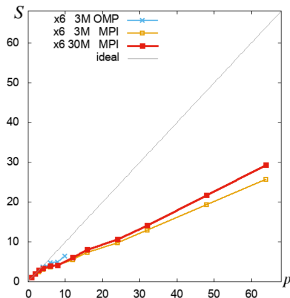


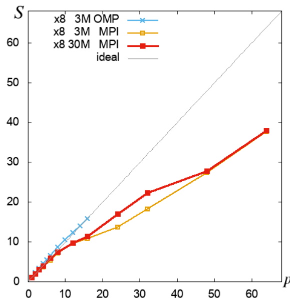


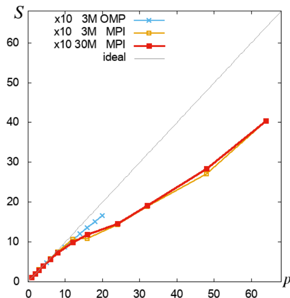


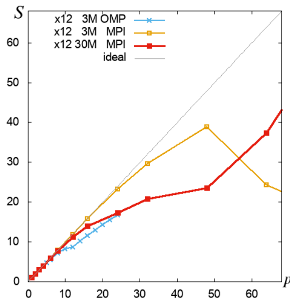


Comparison: OMP vs. MPI norm: $\|\vec{x}\|$

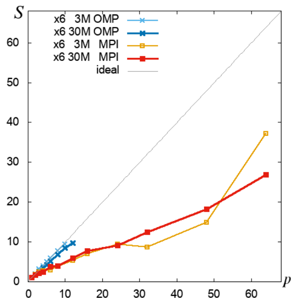


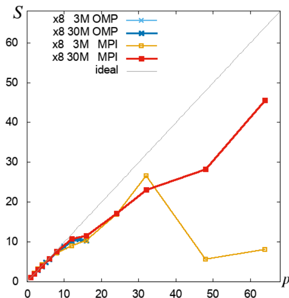


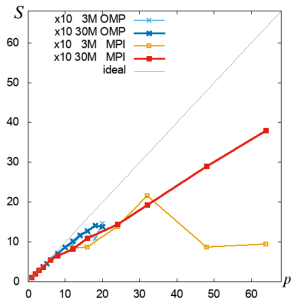


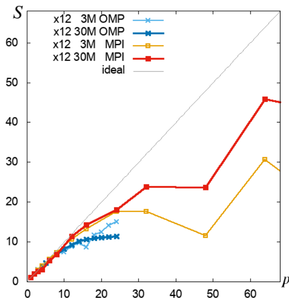


Comparison: OMP vs. MPI mvm: $y = A \cdot x$

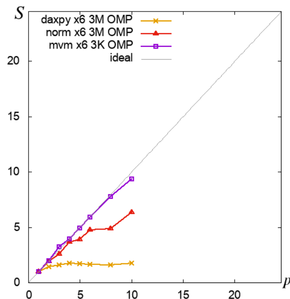


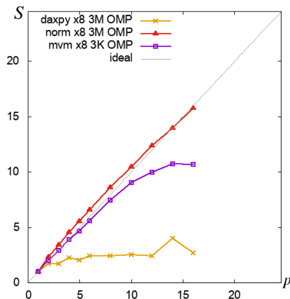


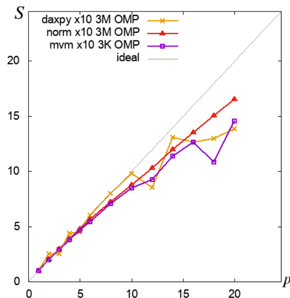


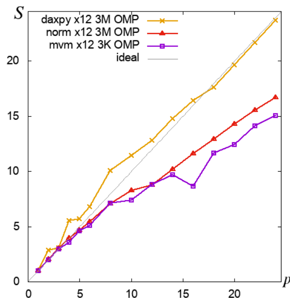


Comparison on OMP: operations (daxpy, norm, mvm)

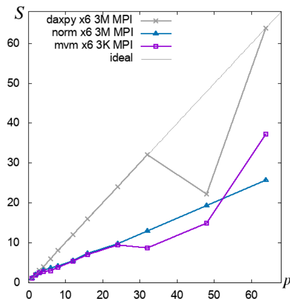


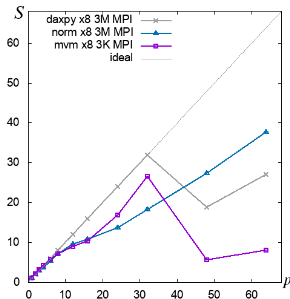


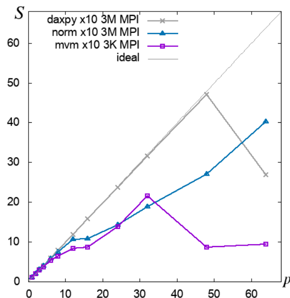


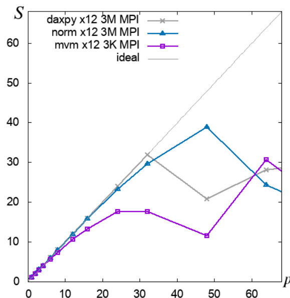


Comparison on MPI (small): operations (daxpy, norm, mvm)

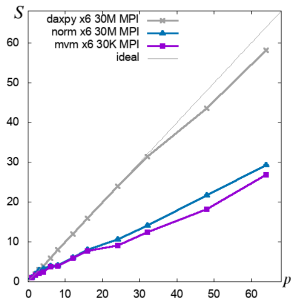


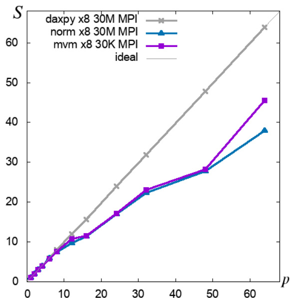


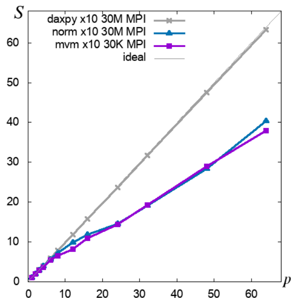


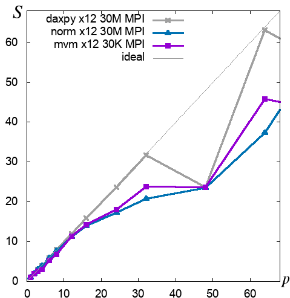


Comparison on MPI (large): operations (daxpy, norm, mvm)









Linear algebra: Congugate Gradient Method (PCG)

$$r_0 = b - Ax_0, \quad p_0 = Hr_0$$
$$i = 0, 1, \ldots$$
$$\alpha_i = (r_i^T H r_i)/(p_i^T A p_i)$$
$$x_{i+1} = x_i + p_i \alpha_i$$
$$r_{i+1} = r_i - A p_i \alpha_i$$
$$\beta_i = (r_{i+1}^T H r_{i+1})/(r_i^T H r_i)$$
$$p_{i+1} = H r_{i+1} + p_i \beta_i$$

where:

- A is a symmetric positive definite $N \times N$ matrix
- H is the preconditioner
- b is the right-hand side
- x_0 is the initial guess (usually $x_0 = 0$)
- x is the solution to be found

Linear algebra: a model for $\mathrm{IC0} + \mathrm{AS}(0) + \mathrm{PCG}$

- 3 x DAXPY
- 2 x DDOT
- 1 x MVM
- 2 x SOL with a block diagonal triangular matrix

The linear system matrix is obtained from the discretization of the problem:
n – dimension in one direction
$N = n \times n \times n$ – the number of unknowns (the dimension of the entire linear system)
$r = n^2$ – matrix bandwidth
$(2d + 1)$-points d-demensional discretization stencil ($d = 3$)

$$L = L_c/L_a = (p - 1)(r + 2)/((2d + 3)N)$$

$$S = \frac{p}{1 + \tau L} = \frac{p}{1 + \dfrac{\tau(p-1)(r+2)}{(2d+3)N}}$$

Linear algebra: PCG experiment

- x6core segment of INM RAS cluster
- $\tau_a = 3.14 \cdot 10^{-10}, \quad \tau_c = 3.06 \cdot 10^{-8}, \quad \tau = \tau_c/\tau_a = 100$
- INMOST software platform: `http://www.inmost.org`
- Test program: solver_test002 (3D: $n = m \times m \times m$)
- $r = m^2$, $n = m^3$, $d = 7$, and $\tau = 100$
- $m = 64, 96, 128, 160, \quad p = 1, 2, 4, 8, 16, 32, 64$
- Linear solvers from PETSc: `https://www.mcs.anl.gov/petsc`

```
-ksp_type cg
-pc_type asm
-pc_asm_overlap 0
-sub_pc_type ilu
-sub_pc_factor_levels 0
```

Speedup: theory and experiment: $\text{IC0} + \text{AS}(0) + \text{PCG}$

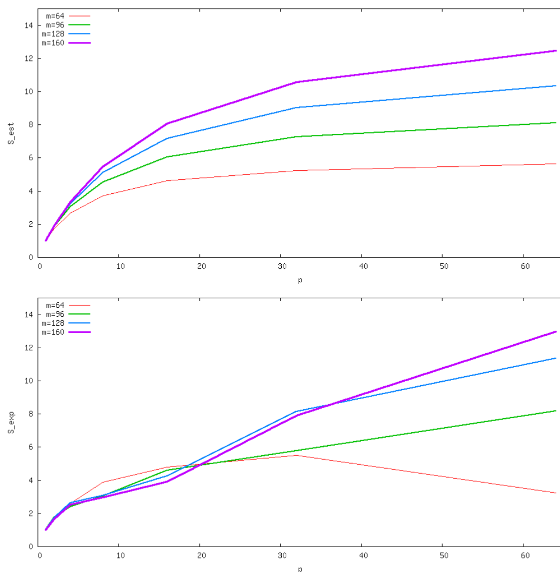


Figure: Speedup (theory and experiment) for $n = 64, 96, 128, 160$

Speedup: theory and experiment: $\text{IC0} + \text{AS}(0) + \text{PCG}$

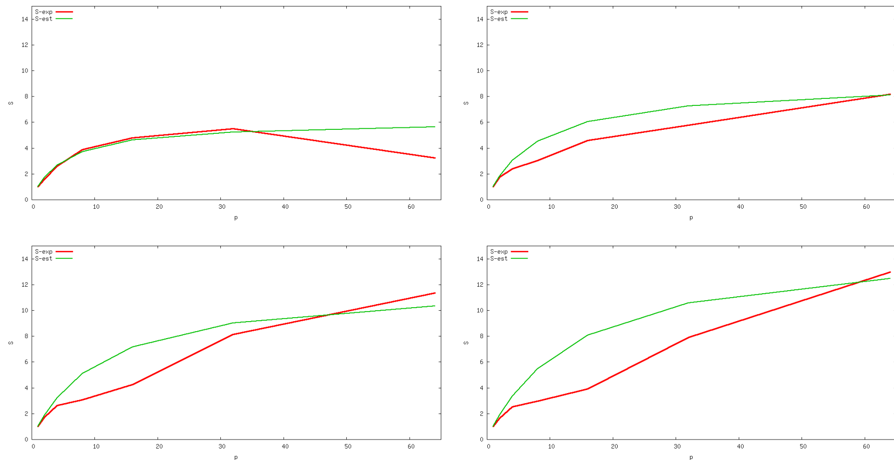


Figure: Speedup (theory and experiment) for $n = 64, 96, 128, 160$

MPI: conclusions

MPI — parallelization of *data*

Main features:

- distributed memory (all Top500 computers are of this type)
- library functions for interprocessor communications
- *more difficult parallelization than by OpenMP*