1 Introduction and overview

- we need high-performance computer systems to solve complex problems
  - climate and environment research
    - ♦ solving models for global and local weather forecasting
    - ♦ climate changes (global warming, ozone hole, …)
    - ♦ forecast for earthquakes, seismic sea waves, …
  - development of new serums or medicaments
  - DNA sequencing
  - analysis of polymers to synthesize new materials
  - performing / evaluating crash tests
  - simulation of turbulent flows and wind tunnel simulations
  - multimedia computing (computer animated movies, …)
  - managing data and processing queries in search engines (Bing, DuckDuckGo, Google, Yahoo, …)
  - …

- example: weather forecasting
  - diameter of the earth: 12 756 km
  - spherical surface: $4 \pi r^2 \Rightarrow$ surface of earth: $5.11 * 10^8$ km^2
  - divide the surface into 2D cells of size 10 km x 10 km
  - consider 20 layers of cells
    - $\Rightarrow$ calculations for about $1.02 * 10^8$ cells (grid points)
    - $\Rightarrow$ assume 5000 floating-point operations for each time step and cell to calculate the temperature, pressure, humidity, etc.
    - $\Rightarrow$ about $5.11 * 10^{11}$ floating-point operations for each time step
    - $\Rightarrow$ weather forecasting for 3 days with 1 minute time steps (4320 intervals) requires about $2.21 * 10^{15}$ floating-point operations in total
  - a computer with 1 GFlops (10^9 Flops) needs about 25.5 days and another one with 1 TFlops (10^{12} Flops) about 37 minutes for this task

    (The above model is an oversimplification of any real model with the only purpose to show the need for high-performance computer systems. The *German Weather Forecast* (Deutscher Wetterdienst) uses three different models to predict the weather (March 2016). The global model uses triangulation (equilateral triangles) of the earth with a grid length of 13 km on 90 vertical layers ($265 * 10^6$ grid points), the regional model for Europe uses grid squares with a grid length of 7 km on 40 layers ($17.5 * 10^6$ grid points), and the regional model for Germany uses grid squares with a grid length of 2.8 km on 50 layers ($9.7 * 10^6$ grid points). The global model uses 130 seconds time steps and predicts the weather for 7 days, the European model uses 40 seconds time steps and predicts the weather for 3 days, and the model for Germany uses 30 seconds time steps and predicts the weather for 18 hours. The model for Germany receives lateral boundary values for its computation from the forecasts of the European model which itself receives its lateral boundary values from the forecasts of the global model. Furthermore the model for Germany receives radar data for convective cells (in German: Gewitterzellen) which are too small for the other models. The computation takes 8 minutes for each day in the global model and 20 minutes for each day in the European model. Weather is chaotic. Therefore the regional model computes 20 ensembles with slightly different boundary conditions, so that it is possible to compute the probability for certain weather conditions (since May 2012). The models are computed on two supercomputers Cray XC40 (each with 17.648 cores, 78 TB main memory, and a floating-point performance of 560 TFlops/s (http://www.dwd.de/DE/derdwd/it/it_node.html). Additionally two Linux-Clusters (with 616/480 Intel Xeon cores and 5/4 TB main memory) serve as frontend systems to do the preprocessing of measured data and the post-processing of the computation (building maps, generating storm warnings, etc.). Furthermore the system contains storage systems with 3.3 PB global disk storage, ten data base servers with 3 PB disk storage for meteorological data, and a tape archive with 60 tape drives and up to 20.000 tape cartridges. You can find additional information in German and English language on the following web pages: http://www.dwd.de/modellierung, http://www.cosmo-model.org.)

- how can we improve the hardware for high-performance computing?
  - new processors with higher and even higher clock rates
  - using instruction pipelines, branch prediction, out-of-order execution, many execution units per processor / core, etc.
  - multiprocessor / multi-core processor systems
  - supercomputers with vector units (today: MMX, SSE, AVX, …)
  - supercomputers with massively parallel processors (MPP; computers with some 100 up to some 1000 processors)
  - (heterogeneous) workstation networks for distributed computing (so-called *virtual computers* or *workstation clusters*)
  - …
- which kinds of memory are available?
  - shared memory (suited for threads, OpenMP, …)
  - distributed memory (suited for processes, MPI, …)
  - distributed shared memory (virtual shared memory, (distributed) global address space)
- how can we interconnect processors, memory, and computers?
  - bus
  - crossbar switch
  - network

- which kinds of “networks” are available?
  - Main memory
  - Gigabit-Ethernet
  - InfiniBand
  - Cray Aries Interconnect
  - …

- some network characteristics

technology	bandwidth	latency
main memory	1 - 50 GB/s	< 0.01 µs
Gigabit-Ethernet	1 Gbit/s	8 - 50 µs
10 Gigabit-Ethernet	10 Gbit/s	2.6 – 50 µs
InfiniBand 1X SDR	2 Gbit/s	1 - 2.6 µs
InfiniBand 4X DDR	16 Gbit/s	1 - 2.6 µs
InfiniBand 12X QDR	96 Gbit/s	1 - 2.6 µs

(**InfiniBand nX**: InfiniBand can aggregate “links” in units of 4 or 12 to increase the bandwidth.

bandwidth: rate at which data can be moved between two points.

latency: minimum time to send a message from one point to another, i.e., the sum of the sender overhead, the receiver overhead, and the signal propagation delay, which is the time for the first bit of the message to arrive at the receiver.

memory bandwidth = base_DRAM_clock_frequency * number_of_lines_per_clock * memory_bus_interface_width_in_bytes * number_of_interfaces

Example: DDR3-1066 or PC3-8500 RAM (different notations with frequency or bandwidth for the same RAM): base clock frequency: 533 MHz, two lines per clock (double data rate) → 1066, 64-bit interface, one memory channel.
Memory bandwidth = 533000000 * 2 * 8 * 1 byte/s = 8528000000 byte/s = 8528 MB/s.
Graphic cards use often a 384-bit data bus (6 memory channels) and GDDR RAMs with higher clock rates, so that they can provide a very high bandwidth.)

- message transmission time is defined by

$$transmission\_time = latency + \frac{message\_size}{bandwidth}$$

- transmission time for small messages limited by latency

- transmission time for large messages limited by bandwidth

- the *effective bandwidth* shows how much of the maximum bandwidth can be utilized and is defined by

$$effective\_bandwidth = \frac{message\_size}{transmission\_time} = \frac{message\_size}{latency + \frac{message\_size}{bandwidth}}$$

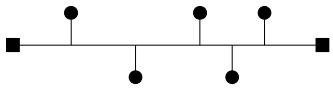
$$= \frac{message\_size * bandwidth}{latency * bandwidth + message\_size} = \frac{bandwidth}{1 + \frac{latency * bandwidth}{message\_size}}$$

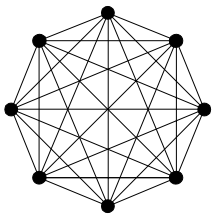
(Without latency the effective bandwidth equals the maximum bandwidth and with latency it depends on the message size. When the message size grows to infinity, the effective bandwidth approaches the network bandwidth and when the message size tends to zero, the effective bandwidth approaches zero as well. If you want to use a network efficiently, you have to use a network with very small latency or you need large messages which then increase the transmission time. On the other hand you must decrease the message size and the number of messages to reduce the communication time in a parallel computation when you want to have an efficient parallel program.)

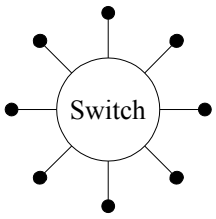
- in general the above formulas are valid for point-to-point communication

(For many nodes it is difficult or nearly impossible to connect every node with every other node (for *n* nodes it would require *n(n-1)/2* connections), so that only a few nodes are connected directly and the transmission of a message from one node to another one will possibly use intermediate nodes. In general, the latency will increase and the bandwidth decrease with the number of intermediate nodes. A routing algorithm must choose an optimal path, if several paths exist between two nodes. The number of intermediate nodes of the shortest route between two nodes defines the *distance* of these two nodes. The largest distance defines the *diameter* of the network.)

- some network topologies
  - bus, complete-connections, star







    - a bus is one of the simplest networks
      - diameter of the network: 1
      - shared medium common to all nodes
      - network costs scale linearly with the number of nodes (mainly cost for bus interface)
      - ideal for broadcasts and similar collective communication operations
      - bounded bandwidth results in poor performance when the number of nodes increases
    - completely-connected network
      - diameter of the network: 1
      - communication between any different pairs of nodes does not interfere with each other
      - very high costs for the interconnection of the nodes

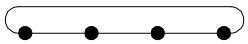
- ♦ star-connected network
  - · diameter of the network: 2
  - · switch simulates a completely-connected network electronically, i.e., disjunct pairs of nodes can communicate with full connection bandwidth

    (Switch provides a bandwidth of $\frac{NumberOfPorts * ConnectionBandwidth}{2}$.

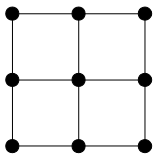
    Even if it provides a lower internal bandwidth, it is a very good solution, because most programs will not use the full connection bandwidth at the same time. It is the preferred topology for switched Ethernet and InfiniBand.)
  - · network costs scale linearly with the number of nodes
  - · sometimes a tree-based hierarchy of switches will be used

– 1D-, 2D-, and 3D meshes (cartesian topologies)

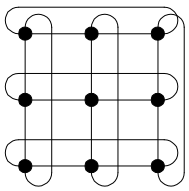
linear array



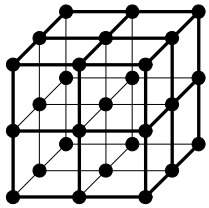
ring



2D mesh



2D mesh with wraparound



3D mesh

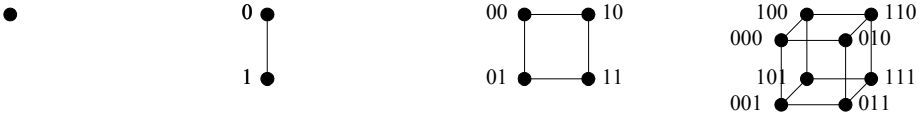
(a *ring* is also called a *1D torus*, a *2D mesh with wraparound* a *2D torus*, etc.)

- ♦ some parameters of these networks for *n* nodes:

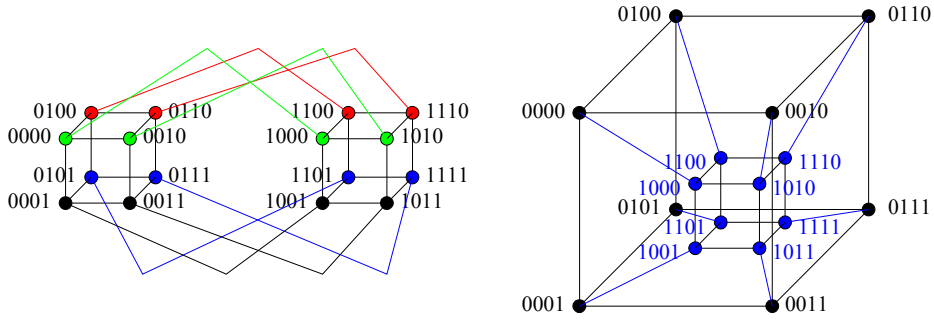
network	diameter	number of connections
linear array	$n-1$	$n-1$
2D mesh	$2(\sqrt{n}-1)$	$2(n-\sqrt{n})$
3D mesh	$3(\sqrt[3]{n}-1)$	$3(n-(\sqrt[3]{n})^2)$
1D torus	$\lfloor n/2 \rfloor$	n
2D torus	$2\lfloor \sqrt{n}/2 \rfloor$	$2n$
3D torus	$3\lfloor \sqrt[3]{n}/2 \rfloor$	$3n$

- ♦ 1D, 2D, and 3D meshes with or without wraparounds are often used because many algorithms naturally map to them

– hypercube



0D hypercube 1D hypercube 2D hypercube 3D hypercube



4D hypercube construction 4D hypercube

- any d-dimensional hypercube is constructed from two (d-1)-dimensional hypercubes by connecting corresponding nodes
- the number of different bits in two node labels shows the minimum distance between both nodes
- diameter of the network: $\log n$ (4 for 16 nodes)

  (Hypercubes are very interesting for larger networks because the diameter increases only logarithmically. The diameter for 64 (4096) nodes for a 2D mesh is 14 (126) and for a 3D mesh it is 9 (45), for a 2D torus it is 8 (64) and for a 3D torus it is 6 (24), while it is 6 (12) for a hypercube. The pay for a small diameter is an increased number of connections.)

- number of connections: $\frac{n}{2}\log n$ (32 for 16 nodes)

  (Number of connections for 64 (4096) nodes: 2D mesh: 112 (8064), 3D mesh: 144 (11520), hypercube: 192 (24576).)

- LinuxLab: available compilers on *pc01* to *pc25* (Linux or Windows (dual-booting), NVIDIA GPU) and on the server *exin* (Linux, doesn't have a NVIDIA GPU)
  - GNU compiler 7.x (Linux and Windows, gcc)
  - Intel Parallel Studio XE 2017 compiler (Linux, icc)
  - LLVM compiler 5.x (Linux, clang)
  - Microsoft Visual Studio 2017 (Windows, cl)
  - NVIDIA CUDA 8.0 SDK (Linux and Windows, nvcc)
  - Oracle Developer Studio 12.x compiler (Linux, cc)
  - Portland Group compiler (Community Edition 2016, Linux, pgcc)

- MacLab: available compilers
  - GNU compiler (gcc-6)
  - LLVM compiler (clang, gcc)

Exercise 1-1:

Implement a program which multiplies two matrices a[P][Q] and b[Q][R]. Remember that you compute the elements of the result matrix c[P][R] with the following formula:

$$c[i][j] = \sum_{k=0}^{Q-1} a[i][k] * b[k][j]$$

Initialize matrices *a* and *b* in an appropriate way and print the values of all three matrices for P = 4, Q = 6 and R = 8, so that you can manually check the results.

Exercise 1-2:

The Italian mathematician *Leonardo da Pisa* (also named *Leonardo Fibonacci*) considered the growth of an idealized rabbit population, which he described with the following function.

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-1} + F_{n-2}, n \geq 2$

Implement a program which computes the sum of the first *n* Fibonacci numbers using at first an iterative (long long fibonacci_iterative (int n)) and then a recursive algorithm (long long fibonacci_recursive (int n)). How long does it take to compute the sums for $35 \leq n \leq 45$? Use function "clock ()" to measure the used CPU time and function "time ()" to measure the elapsed wall clock time.

Exercise 1-3:

Implement a program which sorts the elements of an array with different algorithms. Initialize the array with a random number generator and use the algorithms "bubblesort" and "selection sort" to sort the array. Don't forget to copy the array so that both algorithms use the same values. How long does it take to sort an array with 50.000 and 100.000 integer values? The program should produce something like the following output.

```
Algorithm        array size      CPU time    elapsed time
                                (in seconds)  (in seconds)
Bubblesort          50000          4.48          5.00
Bubblesort         100000         17.95         18.00
Selection sort      50000          2.20          2.00
Selection sort     100000          8.77          9.00
```

Exercise 1-4:

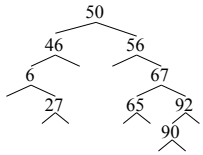
Implement a binary tree for integer numbers with the following functions:

- struct node *insert_node (struct node *root, int data)
- void print_tree_pre_order (struct node *root)
- void print_tree_in_order (struct node *root)
- void print_tree_post_order (struct node *root)
- void delete_tree (struct node **root)

Initialize the tree with 10 random numbers, print the values of the tree in pre-order, in-order, and post-order form. Delete the tree afterwards to free all allocated memory. "delete_tree" needs the address of the pointer to "root" as parameter. "insert_node" returns a pointer to a new node to

support recursive usage or a pointer to “root”, if it is the first node or if “data” is already available in the tree. Possible output with tree structure:

```
Sequence of numbers added to the binary tree.
50 56 46 6 27 67 65 92 90 27

Printing all values in-order. Root = 50
((((()6(()27()))46()))50(()56((()65())67((()90())92()))))
```



Exercise 1-5:

How many primes, Mersenne numbers, Mersenne primes, and perfect numbers are there?

A Mersenne number is a number $M_p = 2^p - 1$, where p is prime. If M_p itself is prime, then it is called a Mersenne prime. P is a perfect number if the sum of all divisors (not including P itself) equals P. Each Mersenne prime has a companion perfect number $P = M_p 2^{p-1}$.

prime number	Mersenne number	Mersenne prime	perfect number
(1)	(1)	(1)	-
2	3	3	6
3	7	7	28
5	31	31	496
7	127	127	8128
11	2047	...	...
13	...	...	...

Implement a program which computes all prime numbers, Mersenne numbers, Mersenne primes, and perfect numbers for a specified intervall. Use the interval [1, 1000000] at first. How long does it take to determine how many numbers? How many numbers of each kind are in the whole interval? How much does the position of an interval influence the computing time (do the computations once more with intervals [1000001, 2000000], [2000001, 3000000], etc.)?

Exercise 1-6:

Stable Marriage Problem. Let *Men* and *Woman* each be arrays of *n* processes. Each man ranks the women from *1* to *n* and each woman ranks the men from *1* to *n*. A pairing is a one-to-one correspondence of men and women. A pairing is stable, if for two men m_1 and m_2 and their paired women w_1 and w_2, both of the following conditions are satisfied:

1. m_1 ranks w_1 higher than w_2 or w_2 ranks m_2 higher than m_1 and
2. m_2 ranks w_2 higher than w_1 or w_1 ranks m_1 higher than m_2

A solution to the stable marriage problem is a set of *n* stable pairings. Implement a program to solve the stable marriage problem. The men should propose and the women should listen. A woman has to accept the first proposal she gets, because perhaps no better one might come along. She can dump the first man if she later gets a better proposal. Animate your program so that you can watch all actions in real time. The processes should execute slowly enough to observe their actions.

1.1 Processor and computer architecture

- Flynn's taxonomy
  - Single Instruction Single Data (SISD)

    Scalar monoprocessor system performing one instruction at a time on a single data item
  - Single Instruction Multiple Data (SIMD)

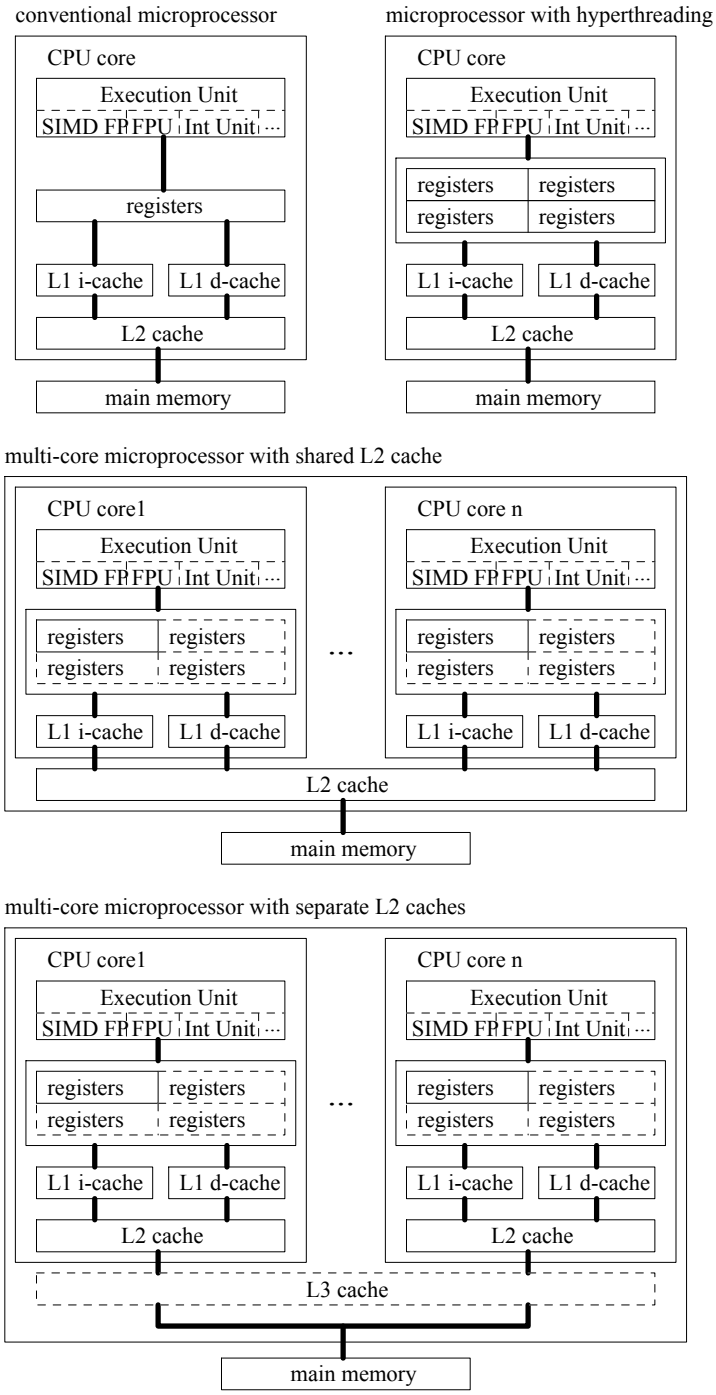
    Vectorprocessor system performing one instruction at a time on multiple data items
  - Multiple Instruction Single Data (MISD)

    There are no real systems available for this class.
  - Multiple Instruction Multiple Data (MIMD)

    (Small) number of independent processors capable of executing individual instruction streams (possibly each processor executes a different program) on multiple data items. This class contains "shared memory multiprocessors", "distributed memory multiprocessors", and "distributed shared (virtual shared) memory multiprocessors".

- evolution of high performance computers
  (http://www.top500.org, http://www.green500.org)
  - 1971: first system with a performance of 1 MFlops
    (CDC 7600, 1.24 MFlops)
  - 1986: first system with a performance of 1 GFlops
    (Cray 2, vector supercomputer, 2 GB memory, 1.7 GFlops)
  - 1997: first system with a performance of 1 TFlops
    (*Intel ASCI Red* at Sandia National Laboratories in Albuquerque, mesh-based MIMD massively parallel computer, 7264 processors (initially Pentium Pro, 200 MHz), 1.2 TB memory, 12.5 TB disk storage, 104 racks taking up about 230 m^2, 1.068 TFlops)
  - 2008: first system with a performance of 1 PFlops
    (*IBM Roadrunner* at U.S. Department of Energy's Los Alamos National Laboratories, Blade Center QS22/LS21 Cluster, 12240 Power XCell 8i (3.2 GHz) and 6562 Opteron dual-core (1.8 GHz) processors, Voltaire InfiniBand 4X DDR, also one of the most energy efficient systems with a total power of 2483.47 kW, 103 TB memory, 296 racks taking up about 560 m^2, 1.026 PFlops)

- current processors and chip multiprocessors (CMP)

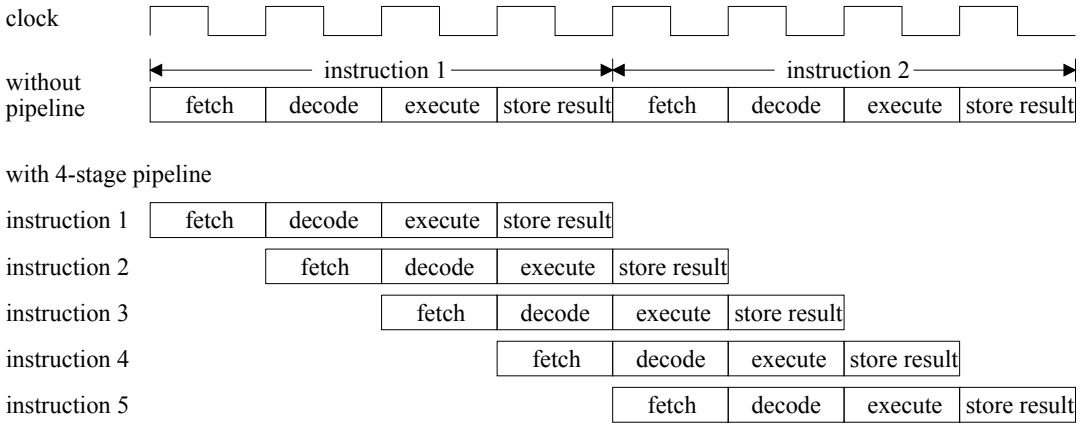


Programs will only benefit from new architectures if the operating system supports them. A processor with *n* register files (hyper-threading technology) can be used like a system with *n* virtual processors. Each logical processor is also provided with its own interrupt mechanism. A processor with hyper-threading technology is not as powerful as a processor with *n* cores because all virtual processors share the execution unit, L1 cache, branch prediction, etc. An Intel P6 processor contains among others two simple instruction decoders, one complex instruction decoder, a retirement unit, a branch target buffer, a memory reorder buffer, a SIMD floating-point execution unit, a floating-point execution unit, two integer execution units, and a memory interface execution unit. Additionally you can connect some multi-core microprocessors with or without hyper-threading technology to “real” multiprocessor systems.

- scalar processor
  - simplest class of processors
  - processes one data item at a time

    (typical data items are integers or floating-point numbers)

- superscalar processor
  - uses instruction level parallelism (instruction pipeline)



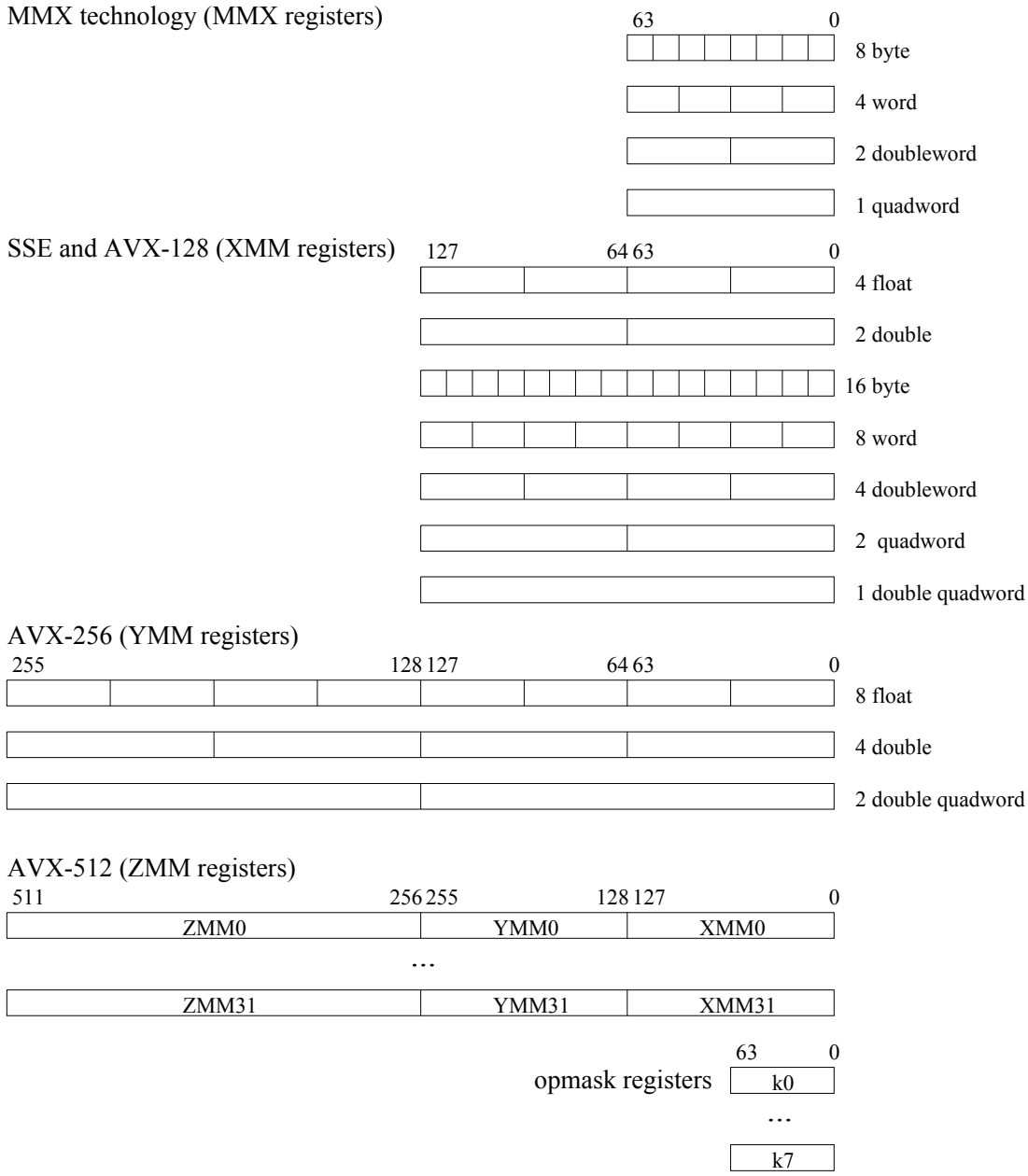
  - dispatches multiple instructions to redundant functional units

    (e.g., instruction decoder, floating-point unit, etc.)

- vector processor
  - operates on one-dimensional arrays of data items (vectors)
  - today even processors for desktop computers or notebooks support vector processing

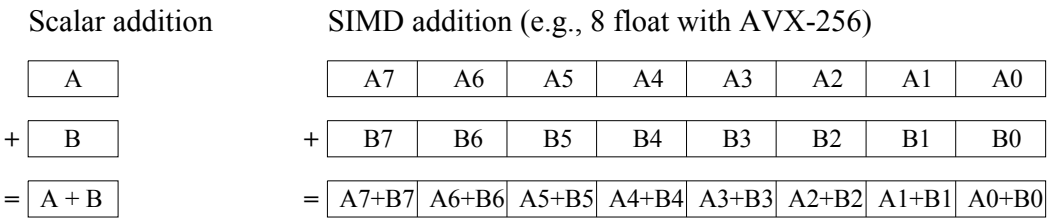
    (Intel *Pentium MMX* (1996) contained eight 64-bit MMX registers and MMX instructions could operate on byte, word, and doubleword integers. Intel *Pentium III* (1999) added eight 128-bit XMM registers and instructions to operate on four single-precision floating-point operands and supported quadword integer operands. *Streaming SIMD Extensions 2* (SSE2) supported double-precision floating-point operands. SSE3, SSSE3, SSE4.1, and SSE4.2 added more operations but did not change the number and size of the registers. The second generation of *Intel Core Processors* (Sandy Bridge, 2011) supports AVX with sixteen 256-bit YMM registers for single and double-precision floating-point operands and many new operations. The fourth generation of *Intel Core Processors* (Haswell, 2013) supports AVX2, which added more operations and 256 bit integer arithmetic, but didn't change the number and size of the registers. The sixth generation of *Intel Core Processors* (Skylake, 2015) supports AVX3 (AVX-512) with thirtytwo 512-bit ZMM registers for up to eight double-precision floating-point operands, eight 64-bit opmask registers, and many new instructions.)

– MMX, SSE, and AVX data types



(The number of XMM and YMM registers increased to 32 with AVX-512. XMM registers are implemented in the lower 128 bits and YMM registers in the lower 256 bits of the ZMM registers. ZMM registers can contain up to 64 byte values or eight double values. The opmask registers offer one bit for every data unit. Each bit decides, if the operation should take place or not for the corrseponding data unit. The size of the opmask registers must increase if the vector register size increases with future processors. Alternatively the granularity of the smallest data unit supported with opmask register may increase, if the size of these registers stays with 64 bits.)

– scalar versus vector operation (packed operation)

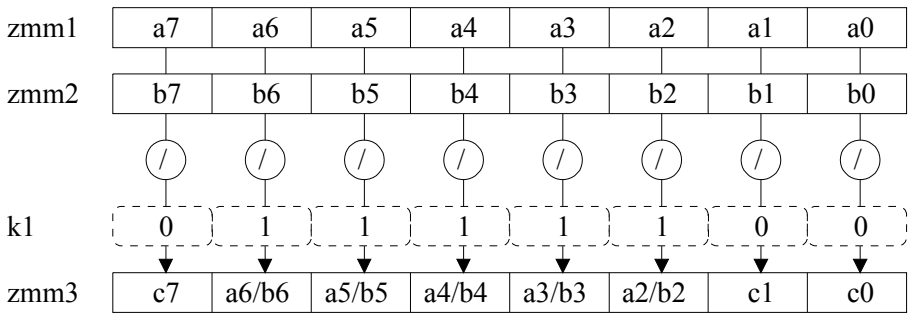


(You can use AVX instructions if your processor supports AVX (e.g., Intel Core i7-2630QM), if your operating system supports AVX (saves and restores AVX registers for every thread or process context switch), and if your compiler supports them as well. The operating systems *Microsoft Windows 7 SP 1*, *Linux 2.6.30*, or *Apple OS X 10.6.8* (Snow Leopard) support AVX. The same is true for the following compilers: *Intel C++ 11.1*, *Microsoft Visual Studio 2010 SP 1*, *GNU gcc 4.4*.)

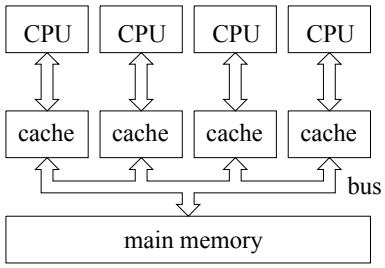
– AVX-512 vector operations with an opmask register

```
double a[8], b[8], c[8];
...
for (int i = 0; i < 8; ++i)
{
  if (b[i] != 0)
  {
    c[i] = a[i] / b[i];
  }
}
```

```
VMOVUPD zmm1, a           /* move unaligned packed double    */
VMOVUPD zmm2, b
VMOVUPD zmm3, c
VXORPD  zmm0, zmm0, zmm0  /* zmm0 = 0                        */
/* comparison predicate 4: zmm0 != zmm2 -> true              */
/*                         zmm0 == zmm2 -> false             */
VCMPPD  k1, zmm0, zmm2, 4
VDIVPD  zmm3 {k1}, zmm1, zmm2
VMOVUPD c, zmm3
```

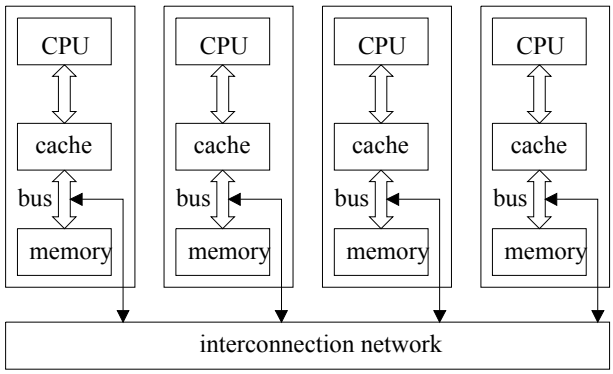


- shared memory multiprocessors



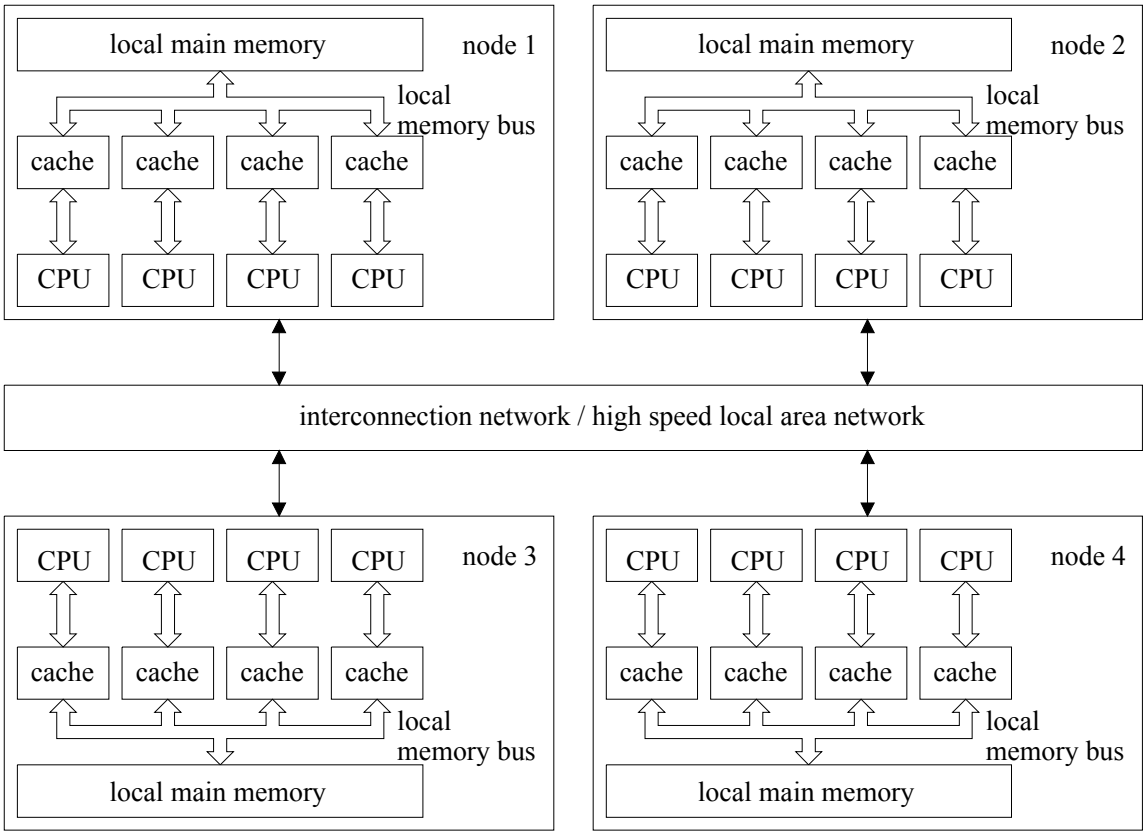
  - this is the default for most multiprocessors
  - the bus / memory can become a bottleneck
  - interprocess communication via main memory possible
  - all processors have the same memory access time to main memory (uniform memory access, UMA)

- distributed (shared) memory multiprocessors



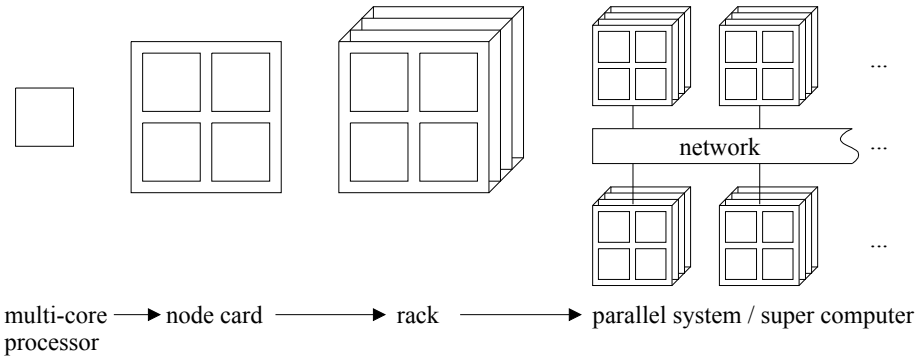
  - e. g., some high performance computers from Silicon Graphics
  - very fast connection network (e. g., *NUMAlink* or *HyperTransport* for AMD-processors)
  - interprocess communication via message-passing
  - shorter access time to local memory than to the memories of the other processors (non-uniform memory access, cache-coherent non-uniform memory access, NUMA, ccNUMA)

- architecture of a local cluster



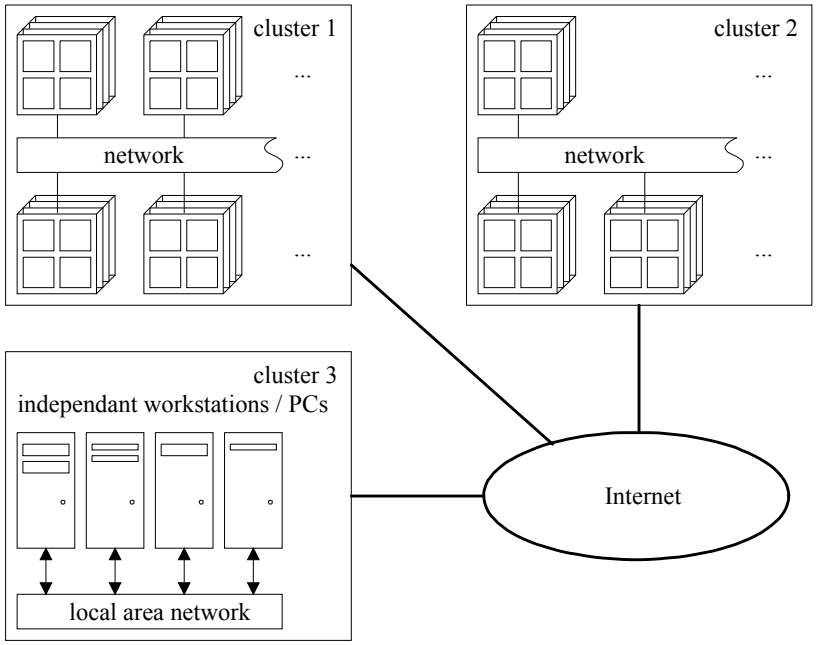
  - the idea is to use (cheap) standard components (off-the-shelf hardware)
  - sometimes called *Cluster Of Workstations* (COW) or *Network Of Workstations* (NOW)
  - most high-end clusters are commercial products with special hardware and software
  - clusters use an optimized network and software environment
  - nodes can be easily added or removed to adjust the computational power of the cluster

- hierarchical parallel system



  - mixes shared and distributed memory
  - this architecture is “standard” for many TOP500 computers

- general architecture of a distributed cluster or grid



  - at large the same as a local cluster but scattered over a wide area
  - many local clusters are interconnected via rather slow Internet
  - such clusters are for example used in grid or particle physics projects or as basis for *cloud computing*

- in general, processors of a cluster are partitioned for different projects

- programs are scheduled on the processors with a resource manager (batch queuing system, job queuing system)

  (e. g., Cluster Resources *Torque/Maui*, Cluster Resources *Moab Workload Manager*, PBS Works *OpenPBS*, PBS Works *PBS Professional*, IBM Tivoli Workload Manager *LoadLeveler*, Oracle *Grid Engine* (former *Sun Grid Engine*), Platform Computing *Load Sharing Facility*, Simple Linux Utility for Resource Management (*SLURM*), etc.)

  – job requirements (number of processors, cpu time, maximum memory, …) are described in a job script file

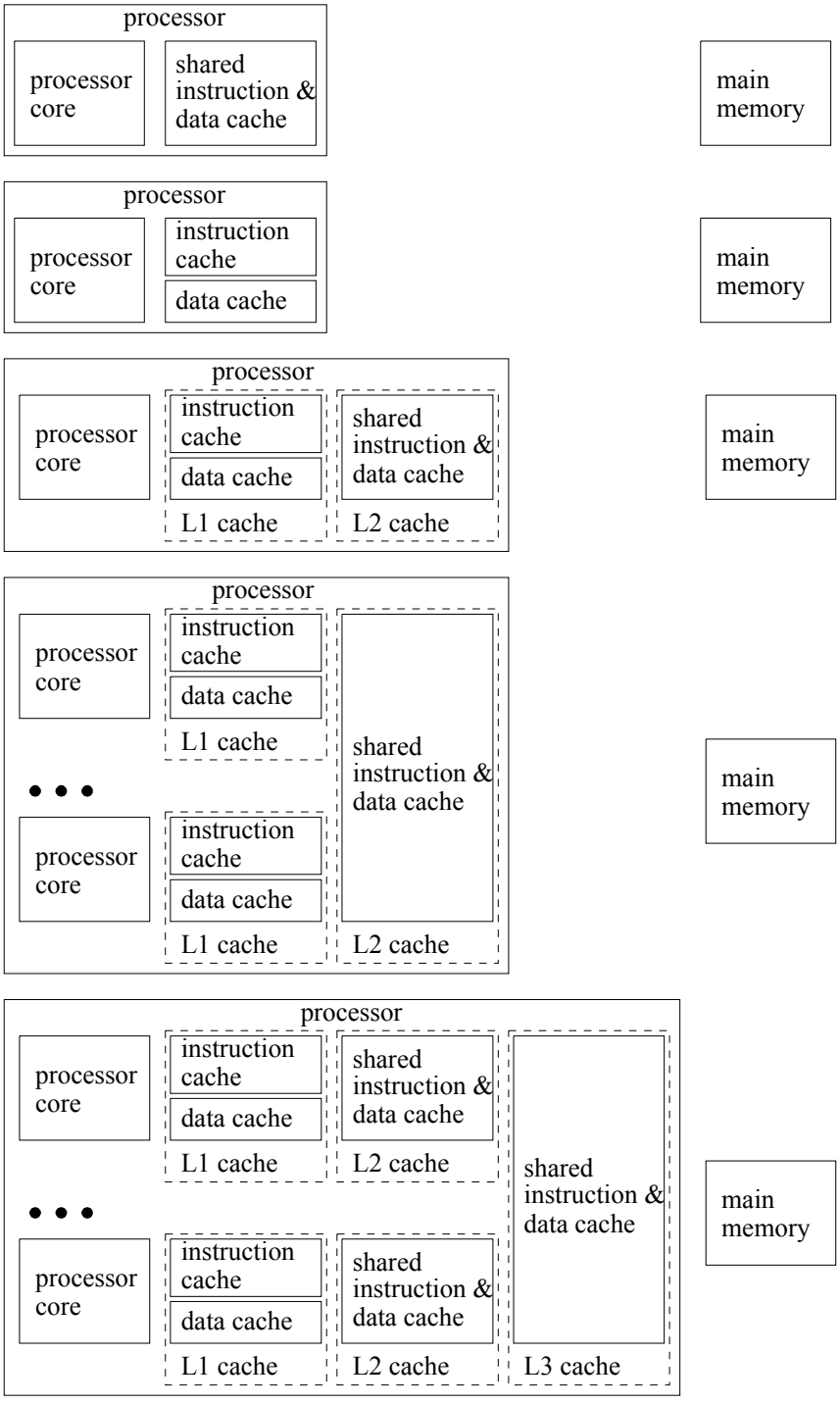
```
#!/bin/sh
#PBS -N job_name
#PBS -l nodes=8:ppn=4
#PBS -l walltime=08:00:00
...
#change to directory from which job was submitted
cd $PBS_O_WORKDIR
#set number of processors to run on
N_NODES=`wc -l < $PBS_NODEFILE`
#run program
mpiexec -np $N_NODES -machinefile $PBS_NODEFILE my_program
```

  – job script file will be submitted to the resource manager

  – resource manager improves overall system efficiency by weighing user requirements against system load

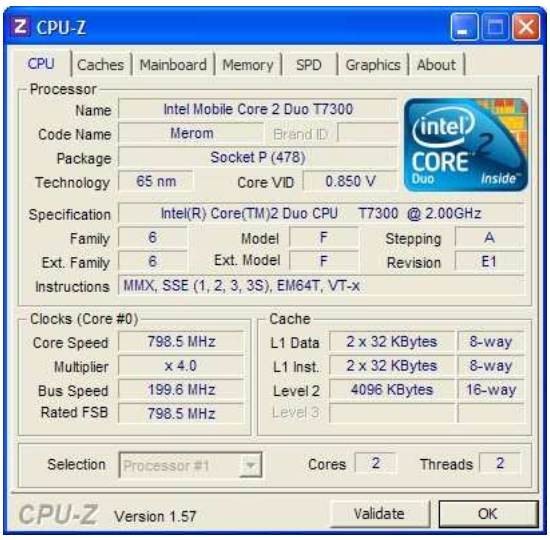
  – resource manager takes care that all job requirements are met when the job starts execution

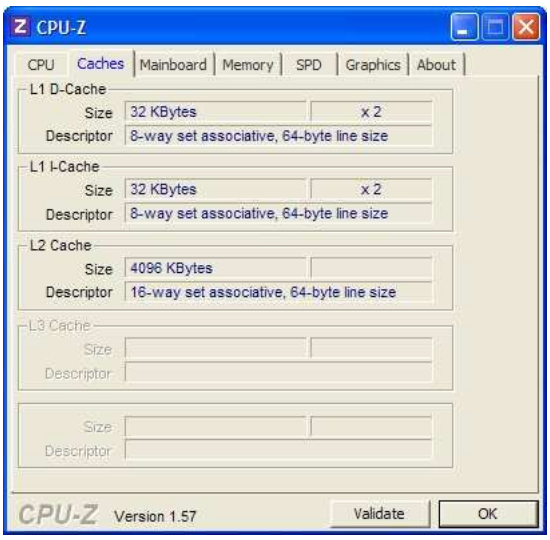
- performance influences of processor caches
  - history of cache hierarchies

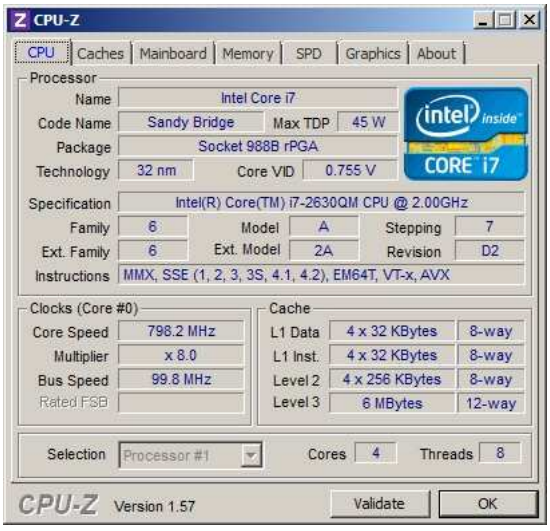


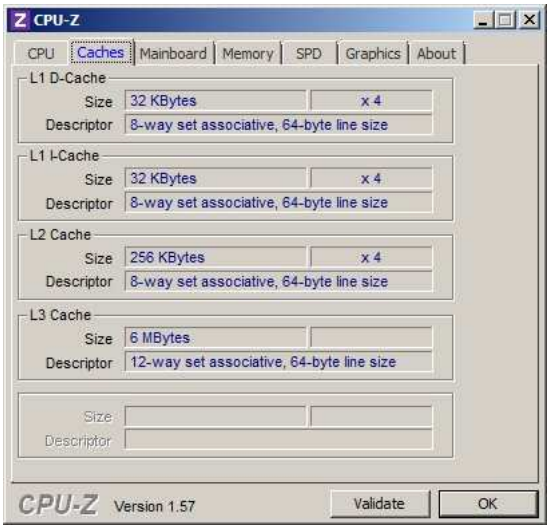
(**Intel 80486**: 8 KB unified instruction and data cache, **IBM/Motorola PowerPC 603**: 8 KB instruction and 8 KB data cache, **Intel Pentium Pro**: 8 KB L1 instruction, 8 KB L1 data, and 256 or 512 KB unified L2 cache. Both Intel 80486 and PowerPC supported an external L2 cache.)

– examples for processor caches







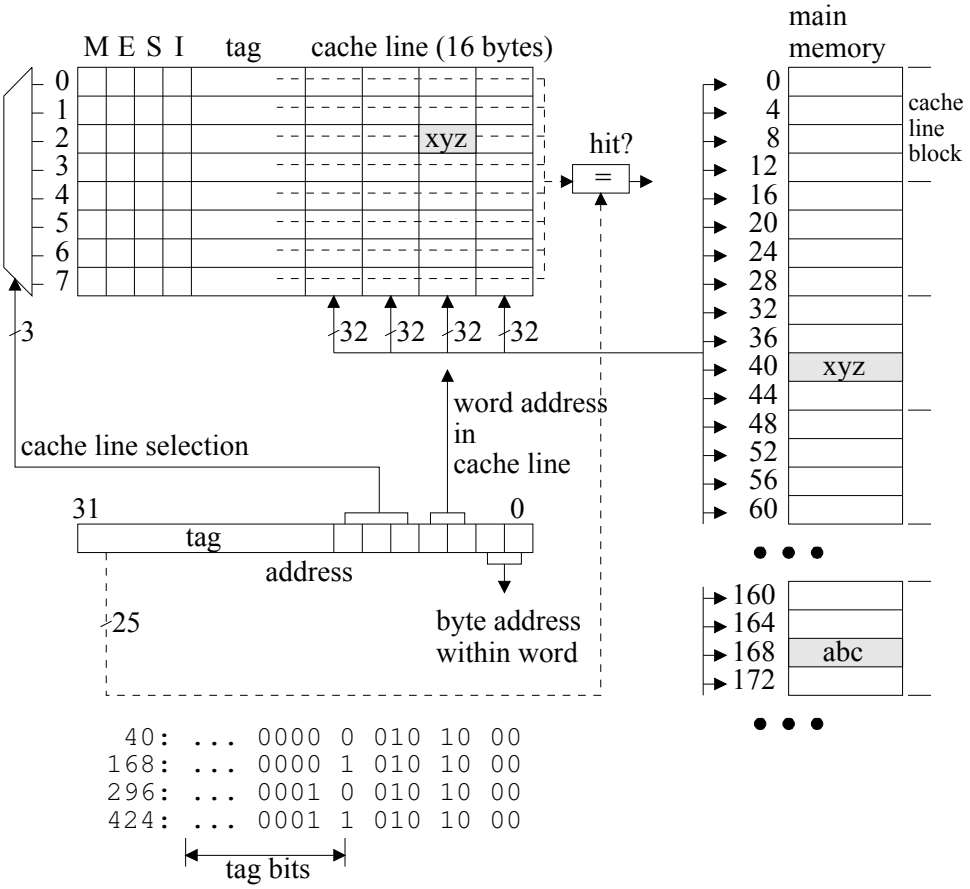


– most interesting is the cache which connects the fast cache hierarchy to slow main memory

Acer TravelMate 6592:	Core 2 Duo T7300,	4 MB L2, 16-way, 64 byte
Fujitsu Celsius H730:	Core i7-4710MQ,	6 MB L3, 12-way, 64 byte
Fujitsu Celsius R940:	Xeon E5-2620v3,	15 MB L3, 20-way, 64 byte
Sun M4000 Server:	Sparc64 VII,	5 MB L2, 10-way, 256 byte

"Solaris 10" for Sparc-processors provides the command "fpversion" to show cache features. Additionally a processor contains translation look-aside buffers (TLB) for address translations. An *Intel Core-i7* contains for example five translation look-aside buffers: an L1 instruction and an L1 data TLB for 4 KB pages, a second level unified TLB for 4 KB pages, and an instruction and a data TLB for large pages.

- cache architectures
  - direct-mapped cache



  - every cache line block from main memory can be stored in exactly one cache line which will be selected via some address bits

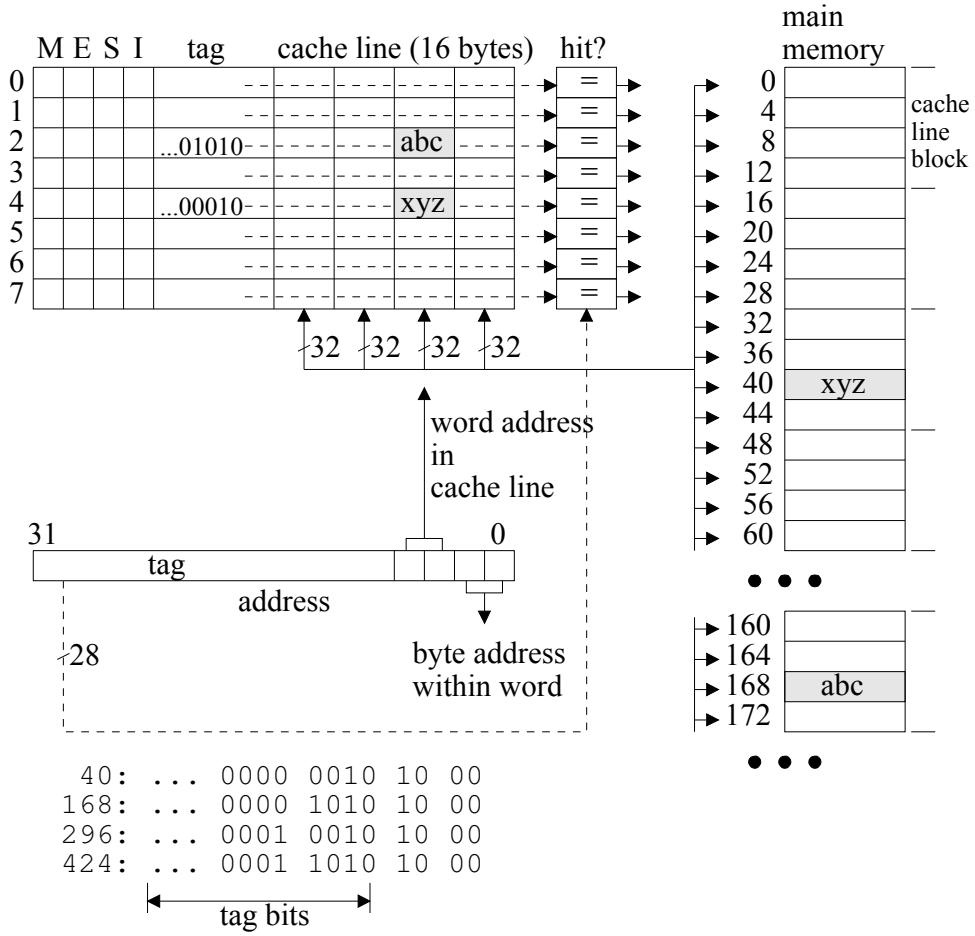
    (simple implementation with low hardware complexity; no replacement algorithm required because of the fixed mapping; hit rate drops sharply if two blocks are used alternately which need the same cache line)

  - a word or byte of the cache line will be selected with other address bits

  - comparing the tag bits of a valid cache line with the higher order bits of the physical address determines if the cache line contains the correct cache line block of main memory

    (M: modified, E: exclusive, S: shared, I: invalid))

– fully associative cache



– every cache line block can be stored in any cache line

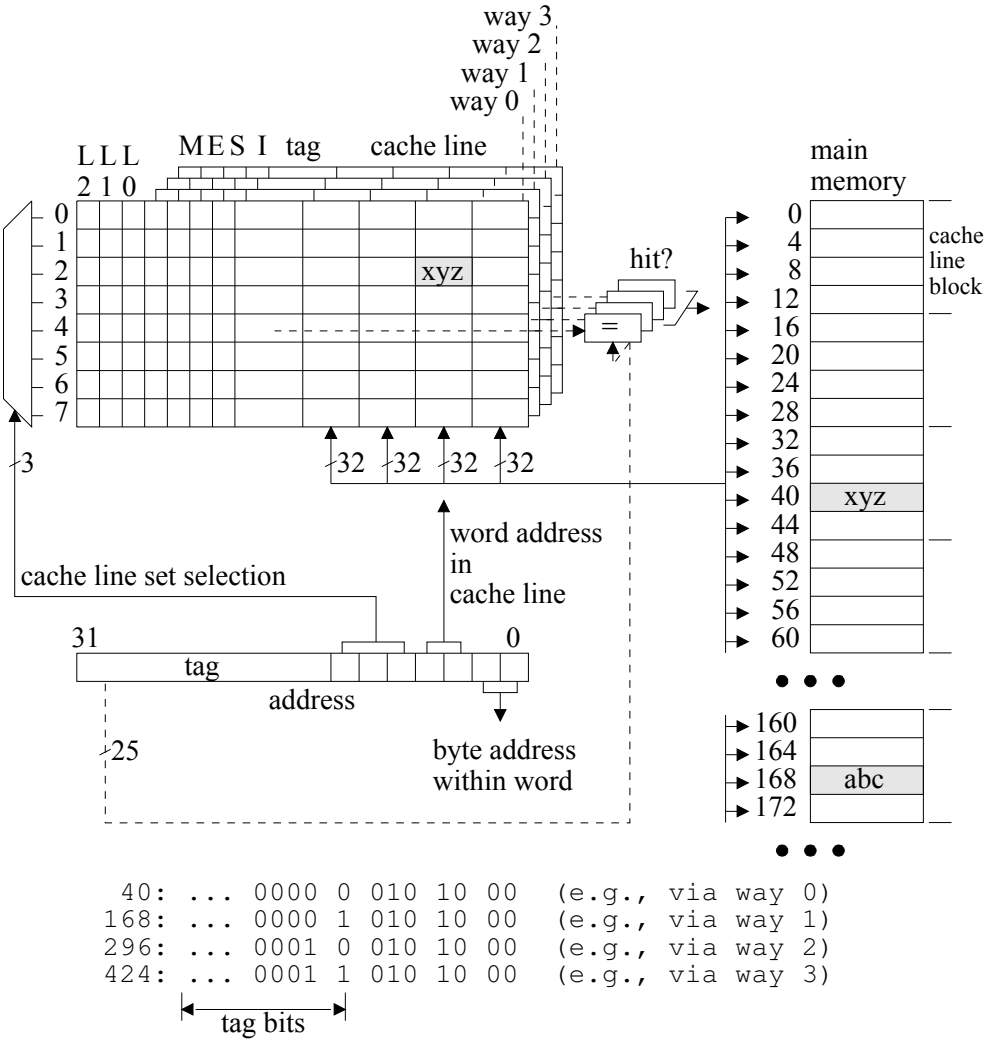
(eliminates the block contention for the same cache line; longer access time than direct-mapped cache due to the associative search; expensive implementation with high hardware complexity as a result of the high number of comparing elements and the implementation of a sophisticated replacement algorithm; in general some kind of LRU (least recently used) algorithm is implemented; best cache organization in consideration of performance)

– comparing the tag bits of all valid cache lines simultaneously with the higher order bits of the physical address determines if a cache line contains the correct cache line block of main memory

(M: modified, E: exclusive, S: shared, I: invalid))

– a word or byte of the cache line will be selected with address bits in case of a *hit*

– n-way set associative cache



– combination of direct-mapped and associative cache

(cache is divided into sets with *n* associative cache lines; a cache line set will be selected via some address bits; economical implementation with middle hardware complexity for the comparing elements and the implementation of a replacement algorithm of a set; in general some kind of LRU (least recently used) is implemented (in our example via bits L0-L2); most commonly used cache organization)

– comparing the tag bits of all valid cache lines of a set simultaneously with the higher order bits of the physical address determines if a cache line contains the correct cache line block of main memory (M: modified, E: exclusive, S: shared, I: invalid))

– a word or byte of the cache line will be selected with address bits in case of a *hit*

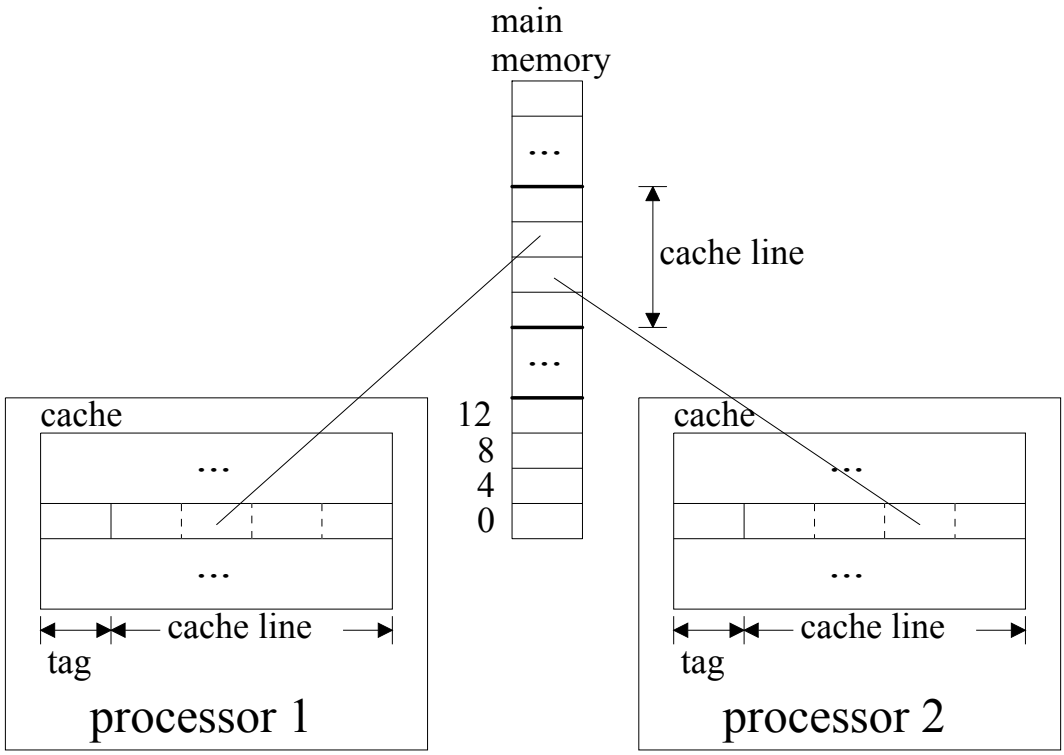
- physical versus virtual addressing
  - in the above examples we have used physical addresses
    - cache access is not possible until the physical address is available (address translation via TLB)
    - cache access time could be significantly reduced if the address translation could be eliminated
    - it would be necessary to use the virtual address for cache accesses
  - using virtual addresses
    - virtual addresses belong to one process
      - the same virtual address can be mapped to different physical addresses in different processes
      - the same physical address can belong to different virtual addresses in different processes in case of a shared memory area
    - somehow it must be guaranteed that each process accesses only its own cache entries
      - the cache can be purged with every context switch
      - an address-space identification can be added to the virtual address which makes it unique

        (Nevertheless it can still happen that the same cache block may exist under different virtual addresses in the cache and thus wasting cache capacity and causing coherence problems (if one block must be invalidated all other copies of the block must be invalidated as well).)

- physical and virtual addresses can be combined
  - physically indexed, physically tagged cache (PIPT)
    - ♦ is simple and in general first choice for large caches
    - ♦ is slow because the virtual address must be translated to a physical address first
  - physically indexed, virtually tagged cache (PIVT) is only of theoretical interest
  - virtually indexed, physically tagged cache (VIPT)
    - ♦ access to a cache line and address translation via TLB can be done simultaneously
    - ♦ coherence problem needs a solution so that this type is only interesting for very small caches (e.g., L1) with low latency
  - virtually indexed, virtually tagged cache (VIVT)
    - ♦ fast lookup
    - ♦ one virtual address in different processes may refer to different physical addresses
    - ♦ different virtual addresses may refer to the same physical address in case of a shared memory area
    - ♦ coherence problem needs a solution

- should we use a processor with shared or separate caches?
  - using a shared cache
    - ♦ may result in contention for the cache
    - ♦ perhaps decreased performance when executing multiple independent sequential programs
    - ♦ parallel programs might benefit because only one program has to fill a cache line and the others can immediately access the data if many processors want to access the same "line" in shared memory

      ⇒ decreasing bus contention
  - using separate independent caches



- ♦ each processor uses its own cache
- ♦ in general increased performance when executing multiple independent sequential programs
- ♦ possibly decreased performance when executing parallel programs
- ♦ if one processor writes to a particular "line" of memory, every cached copy of the old line must be invalidated or updated

  ⇒ a lot of cache and bus traffic (known as *false sharing*)

  ⇒ try to organize data so that such accesses go to different "lines" for each process / thread
- ♦ if a small data structure belongs to two "lines", each access needs an update of two cache lines

  ⇒ try to avoid this problem allocating the data structure at the beginning of a cache line block in memory

- influence of scheduling and processor affinity
  - scheduler attempts to keep each process / thread on the same processor (known as *processor affinity*, *processor binding*)

    ⇒ perhaps the process / thread can benefit from previous cache loads if the cache wasn't purged
  - scheduler does not attempt to keep each process / thread on the same processor

    ⇒ cache must be reloaded (at least the L1 cache)

- influence of page mappings
  - the performance of a process can vary significantly between different runs
    - ♦ the operating system decides which page frame will be used for a page
    - ♦ it may happen that it chooses page frames which collide with each other in a large physically indexed cache
    - ♦ it may also allocate page frames which don't collide in a large physically indexed cache

    ⇒ different cache collision patterns can lead to very large performance differences
  - some operating systems use *page coloring* to avoid such performance problems
    - ♦ sequential pages in virtual memory will be mapped to "sequential" page frames in physical memory
    - ♦ example
      - · consider a direct-mapped cache with a size of 4 MB and a page and page frame size of 8 KB
      - · the cache can hold 4 MB / 8 KB = 512 page frames
      - · if the operating system assigns page 0 to page frame 0 and page 1 to page frame 512 both pages collide in cache
      - · the pages will not collide if the operating systems assigns page 0 to page frame 0 and page 1 to page frame 513 or any other page frame which is not a multiple of 512

- ♦ if a cache can hold *n* page frames the operating system can use the "colors" 0, 1, …, n-1 to color all page frames
- ♦ all page frames with the same color will collide in the cache and all page frames with different colors go to different regions of the cache
- ♦ the operating system must guarantee that pages with color *i* are copied into page frames with color *i*

– now a developer can implement a program which takes care of the cache structure

- ♦ critical data structures will fit into the cache
- ♦ the performance is predictable because sequential pages will be copied into "sequential" page frames

• what else can happen?

– let us try to multiply two matrices in the traditional way (a[P][Q], b[Q][R], c[P][R], $c[i][j] = \sum_{k=0}^{Q-1} a[i][k] * b[k][j]$, 0 ≤ i ≤ P-1, 0 ≤ j ≤ R-1)

$$\begin{pmatrix} c00 & c01 & c02 & ... & c0r \\ c10 & c11 & \boxed{c12} & ... & c1r \\ ... & & & & \\ cp0 & cp1 & cp2 & ... & cpr \end{pmatrix} = \begin{pmatrix} a00 & a01 & a02 & ... & a0q \\ \boxed{a10} & a11 & a12 & ... & a1q \\ ... & & & & \\ ap0 & ap1 & ap2 & ... & apq \end{pmatrix} * \begin{pmatrix} b00 & b01 & \boxed{b02} & ... & b0r \\ b10 & b11 & b12 & ... & b1r \\ ... & & & & \\ bq0 & bq1 & bq2 & ... & bqr \end{pmatrix}$$

– how long will the multiplication of two square matrices take?

number of rows / columns	2040	2048	2056
Intel Core i7-2630QM, 2.0 GHz (64-bit)	95 s	98 s	102 s
Intel Core 2 Duo T7300, 2.0 GHz	78 s	177 s	82 s
AMD Opteron 280, 2.4 GHz	136 s	372 s	137 s
UltraSparc IIIi, 1.6 GHz (64-bit)	14 s	186 s	14 s
Sparc64 VII, 2.4 GHz (64-bit)	9 s	106 s	10 s

– why takes the multiplication of *2048 x 2048 matrices* so long for most of the above processors?

– let us look at the cache hits and misses for the multiplication of *1016 x* 1016 and *1024 x 1024 matrices* on a *Sparc64 VII* processor with *Solaris 10* (mat_mult_cacheline_ijk.c)

1016 x 1016

Cache statistics for matrix	a	b	c
number of cache hits:	1048706056	1015996192	1049772093
minimum hits/set:	487644	472437	487665
maximum hits/set:	520160	503936	520176
number of cache misses:	66040	32775904	32259
minimum misses/set:	30	15240	15
maximum misses/set:	38	16259	16
sets with 0 hit(s):	0	0	0

1024 x 1024

Cache statistics for matrix	a	b	c
number of cache hits:	1072630830	0	1071940111
minimum hits/set:	523280	0	523408
maximum hits/set:	523776	0	523424
number of cache misses:	1110994	1073741824	1801713
minimum misses/set:	512	524288	864
maximum misses/set:	1008	524288	880
sets with 0 hit(s):	0	2048	0

Let us try to explain why we don't have any cache hits for matrix *b* if we use *1024 x 1024 matrices*. The program is implemented in the programming language *C* which stores matrices in row-major order. All matrix elements have the data type *double* (8 byte). Solaris uses 8 KB pages so that the 1024 elements of a row fill one page. The whole matrix is stored in 1024 pages in memory. Let's assume that the matrix starts at address 0 so that the computation of the cache line sets is easy. A cache line is 256 bytes wide (it stores 32 elements of the matrix) so that one page consists of 32 cache lines. A *Sparc64 VII* uses a *10-way L2 cache* with a size of 5 MB so that the cache supports 5 MB / (10 * 256 byte) = 2048 sets.

How will the rows of matrix *b* be mapped into the cache sets?

row	address	cache line set
0	0 - 8191	0 - 31
1	8192 - 16382	32 - 63
2	16384 - 24575	64 - 95
...		
63	516096 - 524287	2016 - 2047
64	524288 - 532479	2048 - 2079 -> 0 - 31
65	532480 - 540671	32 - 63
...		

The first row of matrix *b* will be mapped to the following cache line sets.

address	cache line set
0 - 255	0
256 - 511	1
512 - 767	2
...	
7935 - 8191	31

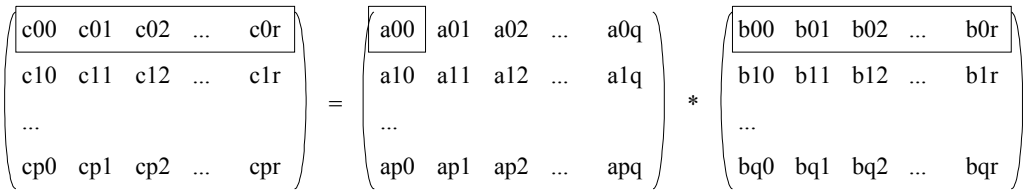
We need one row of matrix *a* and one column of matrix *b* to compute one element of matrix *c*. The 1024 elements of a column of matrix *b* can only use 2048 / 32 = 64 sets of the cache because they are in different rows and each row maps to 32 cache line sets. This means that 1024 / 64 = 16 elements of a column will be mapped to the same cache line. The cache supports only 10 ways so that a cache line will be overwritten before we can reuse the remaining 31 elements of the cache line to compute the next 31 elements of matrix *c*. In the special case that the number of columns is a power of two we have no profit from the cache.

- you can have an unfortunate clash between the cache and layout of data structures
- it is very hard or even impossible to predict if your data structures and the cache structure will work together in the expected way

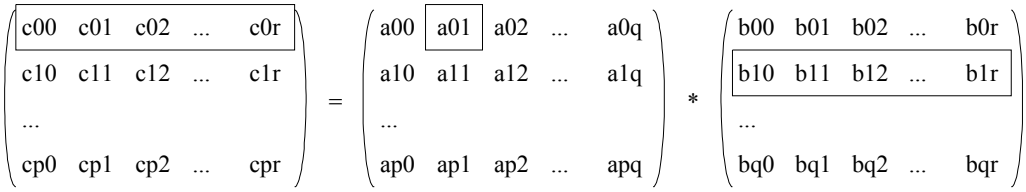
– it may happen that you must adapt your program to another cache structure to get maximum performance

– what can we try to speed up our matrix multiplication if we detect a clash between cache and data structure layout?

♦ change the algorithm to multiply row by row



1st step: c[0][0] = a[0][0] * b[0][0], c[0][1] = a[0][0] * b[0][1], ...



2nd step: c[0][0] += a[0][1] * b[1][0], c[0][1] += a[0][1] * b[1][1], ...

Sparc64 VII, Solaris 10	2040	2048	2056
Sun C 5.13	9 s	107 s	9,5 s
GNU gcc 5.1.0	22 s	25,5 s	22,5 s

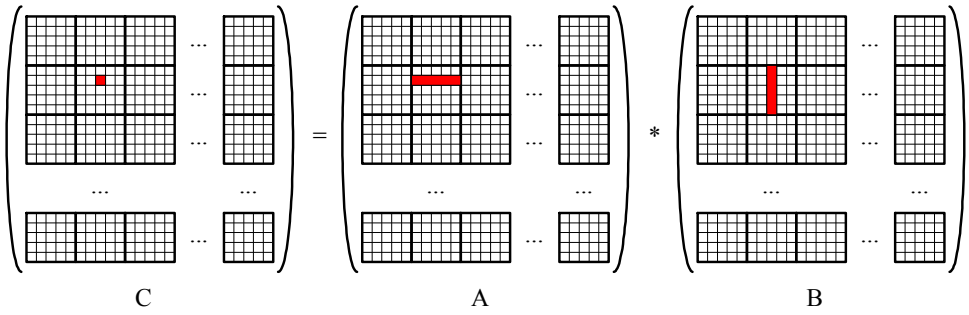
(Program: mat_mult_ikj.c, compiler-options: “cc -std=c11 -m64 -fast -fd -v -xtarget=sparc64vii”, “gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-prototypes”. *Sun C* creates a program for *mat_mult_ikj.c* with roughly the same performance as the program for *mat_mult_ijk.c*. The reason for this behaviour is that the optimizer of the compiler automatically interchanged the middle and inner loop for *mat_mult_ijk.c*, so that it multiplied row by row as well (you can see this optimization, if you add option “-g” to the above command for “cc” and then run “er_src <name of your executable>”, which shows a source code listing combined with many compiler commentaries).

- transpose matrix *b* so that columns turn to rows and we can profit from the cache

Sparc64 VII, Solaris 10	2040	2048	2056
Sun C 5.13	12 s	21 s	13 s
GNU gcc 5.1.0	21 s	25 s	22 s

(Program: mat_mult_transpose_ijk.c, compiler-options: “cc -std=c11 -m64 -fast -fd -v -xtarget=sparc64vii”, “gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-prototypes”. The times above include the time to transpose matrix *b*.)

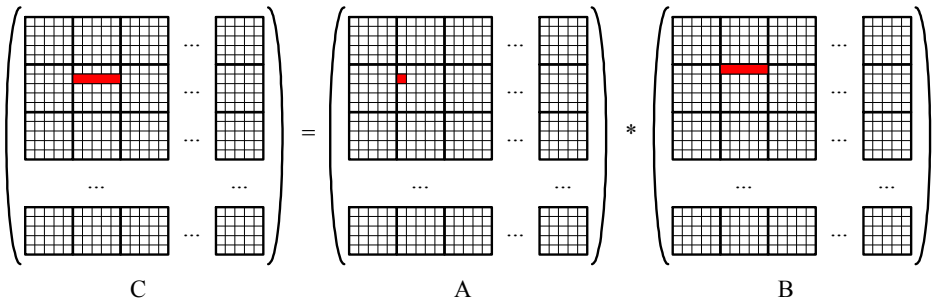
- use submatrices and traditional multiplication



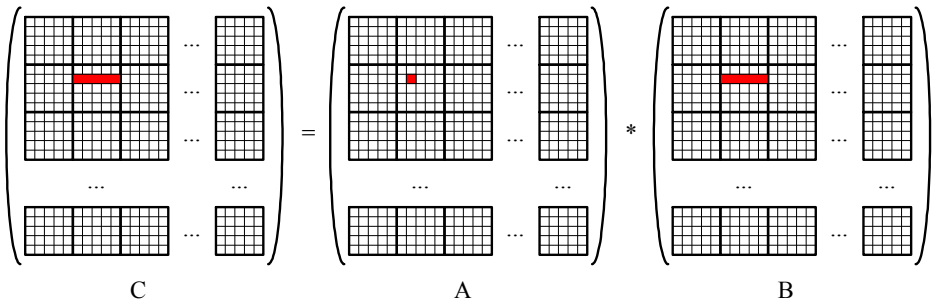
Sparc64 VII, Solaris 10	2040	2048	2056
Sun C 5.13	13 s	206 s	13,5 s
GNU gcc 5.1.0	56 s	186 s	57 s

(Program: mat_mult_block_ijk.c, compiler-options: “cc -std=c11 -m64 -fast -fd -v -xtarget=sparc64vii -DBLOCKSIZE=256”, “gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-prototypes -DBLOCKSIZE=256”. There is probably once more a clash between cache and data structure layout.)

- use submatrices and row by row multiplication



1st step: c[i][j] = a[i][k] * b[k][j], c[i][j+1] = a[i][k] * b[k][j+1]



2nd step: c[i][j] += a[i][k+1] * b[k+1][j], c[i][j+1] += a[i][k+1] * b[k+1][j+1]

Sparc64 VII, Solaris 10	2040	2048	2056
Sun C 5.13	14 s	15 s	14 s
GNU gcc 5.1.0	22 s	26 s	22 s

(Program: mat_mult_block_ikj.c, compiler-options: “cc -std=c11 -m64 -fast -fd -v -xtarget=sparc64vii -DBLOCKSIZE=256”, “gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-prototypes -DBLOCKSIZE=256”.)

- influence of different optimization levels

  (Sparc64 VII, Solaris 10, Sun C 5.12: cc -std=c11 -m64 {-xO2 | -xO3 | -fast} -fd -v {-DP=2040 | -DMY_P=2040} -DQ=2040 -DR=2040 -DBLOCKSIZE=256)

all times in seconds	-xO2			-xO3			-fast		
number of rows / columns	2040	2048	2056	2040	2048	2056	2040	2048	2056
mat_mult_ijk	123	1124	125	21	108	23	12	111	12
mat_mult_ikj	24	26	27	21	108	24	12	108	13
mat_mult_transpose_ijk	57	61	58	48	317	45	12	21	13
mat_mult_block_ijk	61	174	62	23	213	24	20	193	21
mat_mult_block_ikj	27	35	28	16	16	16	15	15	15
mat_mult_gsl_dgemm	31	32	31	30	32	31	31	32	31
mat_mult_sunperf_dgemm	2	2	2	2	2	2	2	2	2

  - a higher optimization level doesn't necessarily result in better performance

    (mat_mult_ikj, mat_mult_transpose_ijk, mat_mult_block_ijk for matrices with 2048 rows / columns)

  - multiplying row by row results in better performance than multiplying elements of a row with elements of a column

    (using the cache more efficiently and possibly avoiding a clash between cache and data structure layout)

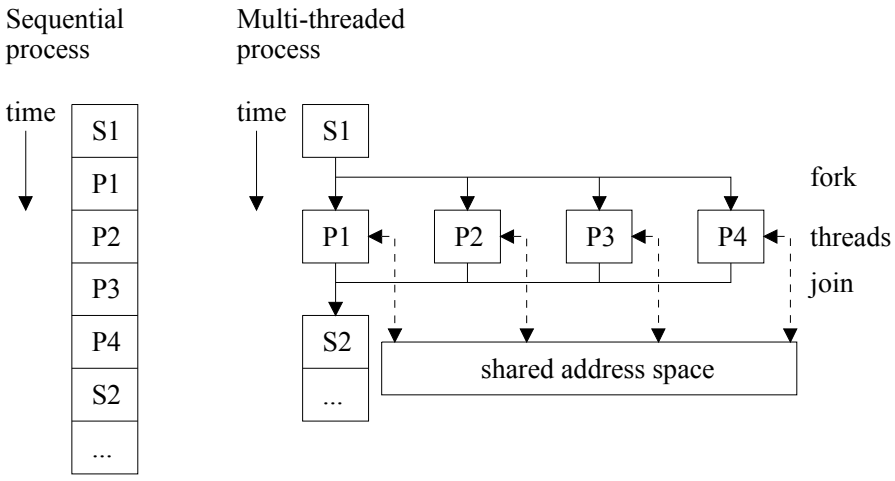
  - often it is more efficient to use optimized (commercial) library functions

    (E.g., functions from the *Performance Library* from *Oracle Developer Studio*. The "dgemm" implementation (Double GEneral Matrix Multiplication) from the *Performance Library* is already parallelized so that it can take advantage of multi-core or multiple processors if you set the environment variable PARALLEL, e.g., "setenv PARALLEL 2" to create two threads.)

1.2 Programming models

- different models are available
  - (distributed) shared memory model
    - ♦ two architectures
      - · memory access time doesn't depend on the memory address or location (uniform memory access, UMA)
      - · memory access time depends on memory location relative to the processor (non-uniform memory access, NUMA, ccNUMA)
    - ♦ automatic parallelization, threads, OpenMP are best suited

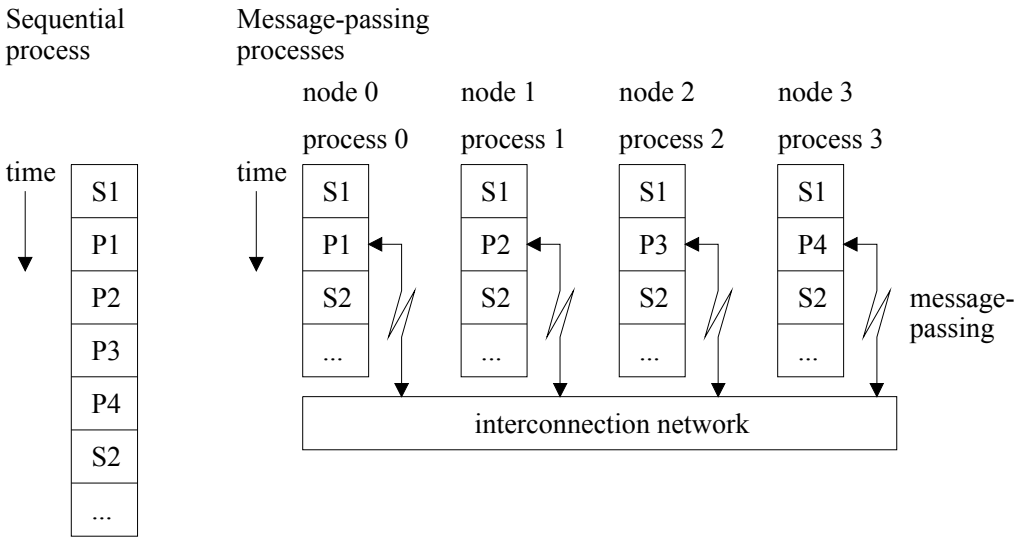
      (Compiler can automatically parallelize certain loops or the programmer adds some directives to the program to support the work of the compiler. In these cases the programmer has nearly no control over the behavior of the threads because everything is done by the compiler. On the other hand the programmer can explicitly implement a multithreaded program, e.g., using the POSIX threads library. When he or she does it well, the program provides good performance because he or she has complete control over the behavior of the program. Sometimes this is also called the *fork-join model*, although no processes are forked. Creating and joining threads and synchronizing access to shared data leads to some overhead.)



    - ♦ *sockets* and *message-passing interface* also possible, but in general not very efficient

– distributed memory model

♦ *sockets* or *message-passing interface* are necessary



♦ in general, compiler cannot automatically parallelize programs working in distributed memory

(For example, it cannot automatically add calls to the message-passing interface library. A programmer must parallelize this class of programs. *High Performance Fortran* can do a few things automatically.)

♦ in general, it takes longer to process each of P1-P4 in a message-passing program than in a sequential program due to

· communication overhead

· synchronization

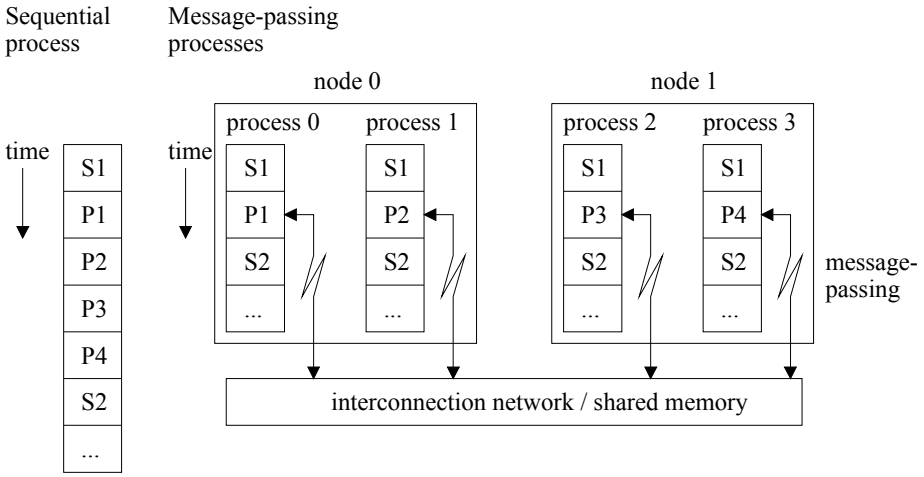
· workload unbalance

– hybrid memory model

♦ combines shared and distributed memory models

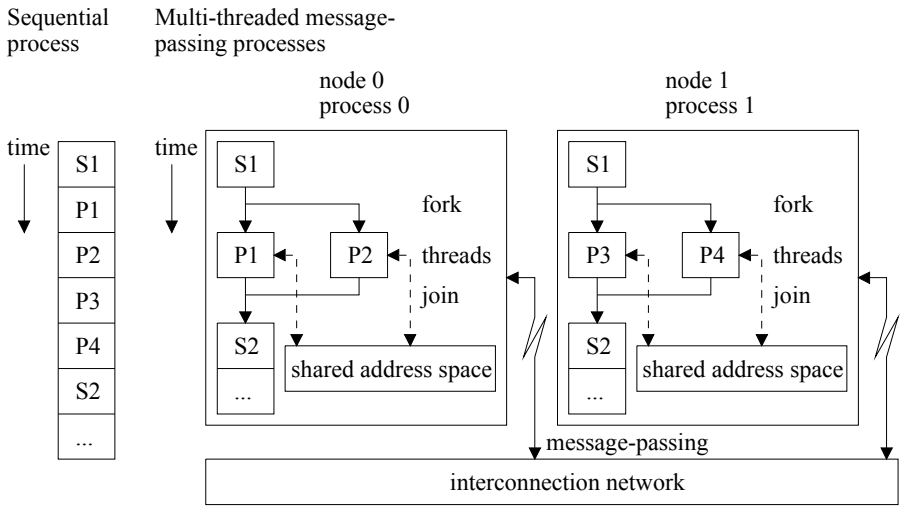
♦ combination of automatic parallelization, threads, OpenMP, sockets, and message-passing interface is possible

♦ multiple single-threaded processes per node



(Some message-passing interface libraries take care of multiple processes per node and use automatically *shared memory* instead of a *network connection* for the interprocess communication of these processes so that the intranode message-passing is optimized due to communication latency and bandwidth. In general, you create one process for every available processor of a node.)

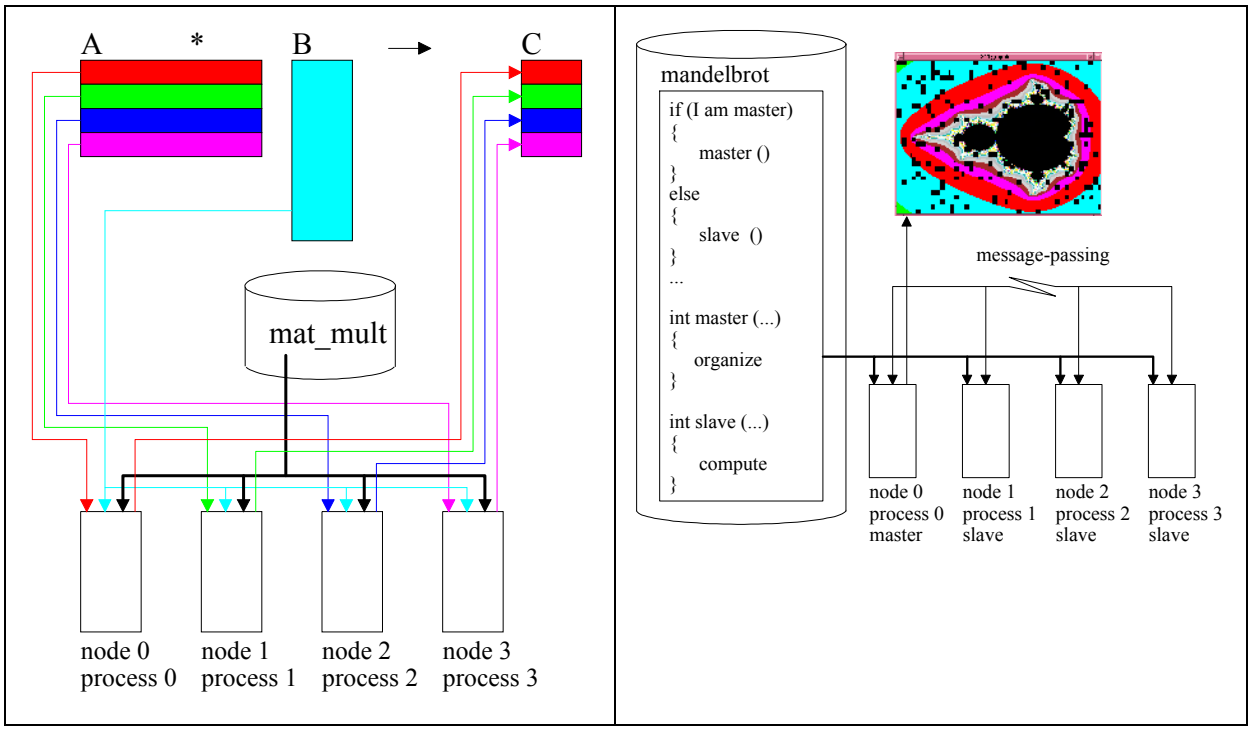
♦ one multithreaded process per node



(Intranode communication uses *shared memory* and internode communication uses *message-passing* so that communication latency and bandwidth is always optimized. The threads can be user-coded or compiler-generated as in the shared memory model.)

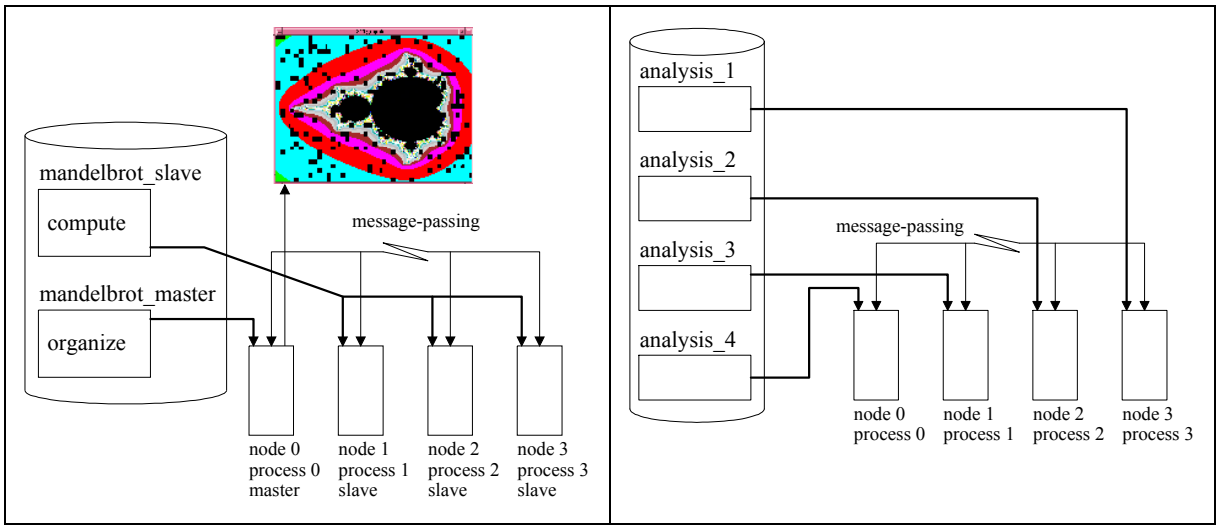
- Single Program Multiple Data (SPMD)
  - high-level programming model which can be built upon any combination of the previously mentioned models
  - all processes execute the same program but not necessarily the same code
  - usually these programs contain a logic which allows different processes to select a portion of the program

    (E.g., in the message-passing interface model each process has a rank (kind of identifier) which can be used to branch to different parts of the code.)
  - all processes may use different data sets

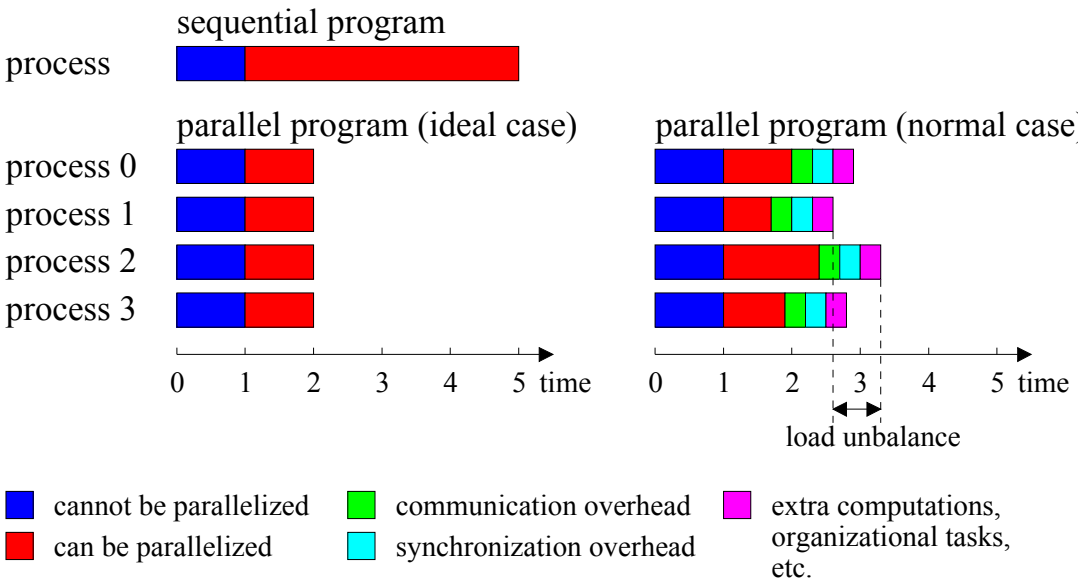


- Multiple Program Multiple Data (MPMD)
  - high-level programming model which can be built upon any combination of the previously mentioned models
  - uses multiple executable programs (different programs for different task)
  - all processes may use different data sets
  - different processes can perform different operations on the same data set
  - all processes work together to solve the same problem

    (In general, they work independently for a long time before they exchange data to proceed to the next step.)



- parallelizing a program



  - execute parallel part in different processes or threads if the program can be parallelized

  - parallelization normally adds costs for
    - communication
    - synchronization
    - extra work

  - in general, the work will not split equally which causes a load unbalance between different processes / threads

  - guidelines
    - increase the fraction of the program that can be parallelized
    - balance the workload of parallel processes
    - minimize communication and synchronization time
    - use compiler support if available

- automatic parallelization
  - user activates this feature with a compiler option

    (compiler analyses a “normal” sequential source code, user has not added any hints for the compiler which parts could be parallelized)
  - compiler performs dependence analysis to determine which parts of the program can be executed in parallel

    (Nothing will be parallelized if the compiler does not find independent parts or if the code is too complex.)
  - normally the compiler focuses on loops and splits their work in disjoint chunks which it assigns to threads
  - success or failure of automatic parallelization depends on
    - the programming language

      (Fortran is easier to analyse at compile time than C or C++ due to their higher abstraction level)
    - the application area

      (Some algorithms are inherently parallel, e.g., matrix multiplication, while it is hard to detect parallel parts in other algorithms or it may even be impossible to find parallel parts because they do not exist.)
    - the coding style

      (The compiler needs to extract as much information as possible from the application and that may be a great challenge or even impossible depending on the coding style.)
    - the compiler

      (It is very complicated to identify parallel parts at compile time so that you need a sophisticated compiler with the latest compiler technology.)

- OpenMP
  - uses compiler directives (#pragma) in the source code
  - programmer defines which parts can be executed in parallel
  - user activates this feature with a compiler option
  - source code itself will not be modified so that the program can still be compiled, even then when the compiler does not support OpenMP

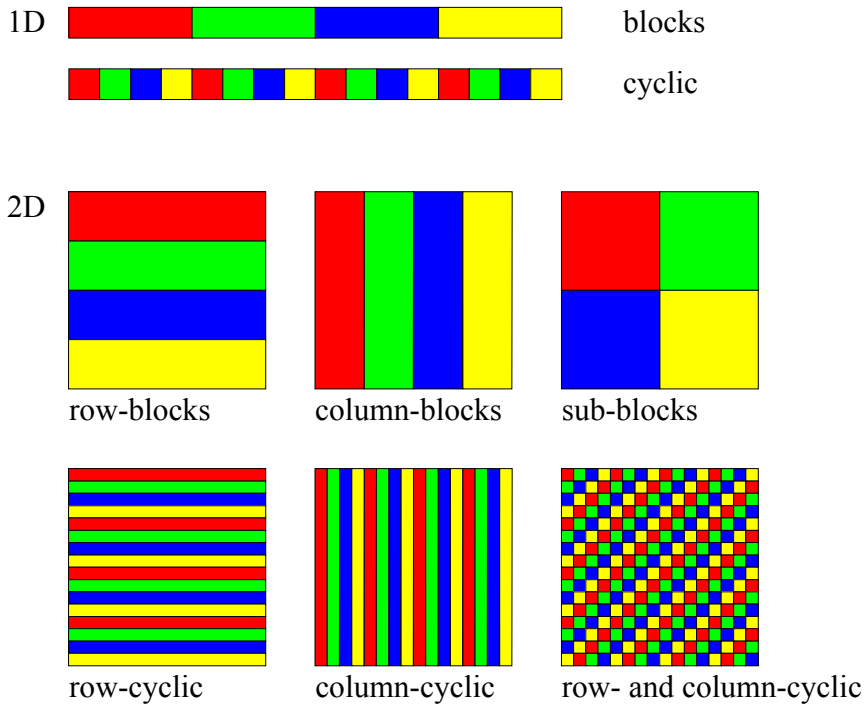
    (In this case, the compiler may warn about an unknown *pragma* and creates a normal sequential program.)
  - OpenMP supports an extensive set of features to specify parallelism, control the workload distribution, and synchronize the threads
- threads
  - explicit parallelization using a special programming interface, e.g., POSIX Threads, Windows Threads, Java Threads, etc.
  - programmer can use functions of a library to implement and control parallelism

    (create a thread, synchronize thread activities, terminate a thread, etc.)
  - possibly the program must be ported to another thread library if the original library is not available on a platform
  - maybe expensive to develop and hard to maintain if the application must be available on incompatible platforms

- sockets
  - not a programming model but a protocol to send and receive packets across a network
  - available for most operating systems
  - very low level
- message-passing interface
  - today's most popular programming model for a cluster of nodes
  - supports point-to-point and collective communication
  - supports blocking and non-blocking operations
  - supports process groups and topologies
  - supports derived data types
  - supports various methods of data partitioning

- common features to shared and distributed memory parallelism
  - some parts must always be carried out serially, leading to overhead in the execution time if the results must be shared
    - you can perform the computation in one process and broadcast the result to all processes
      (overhead for communication)
    - every process / thread can compute the result locally if the data used in the computation is already available
      (overhead for redundant work)
  - sometimes processes / threads must share data
    - processes must actively send and receive the data to be shared in distributed memory systems
      (each send-receive operation is an implicit synchronization between processes)
    - processes / threads can simply access the data in shared memory systems
      (accesses must be synchronized if a process / thread wants to modify data; perhaps they must explicitly synchronize the order of writing and reading a variable)
  - work must be assigned to processes / threads
    - each process / thread works on a chunk of data and executes the same computations on its block
      (data parallelism)
    - each process / thread executes a different portion of code
      (function parallelism)

– data must be partitioned in one or another way



– work can be assigned in a static or dynamic fashion

♦ a static allocation of work is best when

· work can be broken into a number of parts equal to the number of processes / threads (no idle processes / threads)

· execution time of each part is roughly the same

♦ a dynamic allocation of work is best when

· there are more pieces of work than processes / threads

· execution time for each piece is different or even unknown

♦ dynamic allocation results in more overhead but has the benefit of a more load-balanced execution

- differences between shared and distributed memory parallelism
  - memory access from processes / threads
    - processes in a distributed memory system “own” all memory in their virtual address space, i.e., variables can be modified without problems
    - threads in a shared memory system “share” the virtual address space of their process
      - each modification of a shared variable must be synchronized
      - declarations within a function call are automatically local to the thread which calls the function

        (they are part of the individual thread stack and guarantee that threads can run within the same process context)
      - thread-local storage allows a thread a private memory area

        (The difference between declarations within a function and thread-local storage is that data in thread-local storage will persist from one function call to another. This is like a static variable, except that each thread gets an individually addressable copy.)

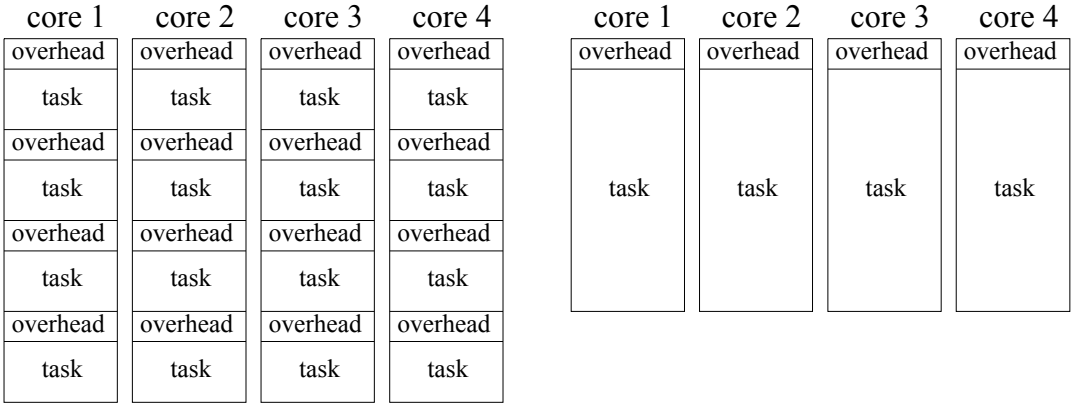
- mutual exclusion (synchronization of memory access)
  - memory accesses in distributed memory do not need a special protection
  - sometimes accesses to shared memory must be protected
  - both reading and writing can be critical

    (A parallel update of a memory location is always critical because the order of the updates will determine the ultimate value stored in the location. It is also critical if one thread reads from and another one writes to a memory location because the reading thread can read the old or the new value. The original serial code normally allowed only one value (read-after-write or write-after-read) so that you must add logic that guarantees that values are read and written at the correct time. It is not critical if many threads read the same memory location.)
  - for many reading threads and one or more writing thread(s) use a *read / write lock* if available

    (Using normal *mutual exclusion* may result in a performance bottleneck if many reading threads (readers) and writing threads (writers) must be synchronized so that an arbitrary number of readers can read a memory location as long as no writer updates this location. On the other hand, a writer must be allowed to update the memory location at least after some time. A *read / write lock* is an appropriate synchronization mechanism if updates of a memory location are much more rarely than reading the location. This mechanism allows many readers to enter the protected area of code (critical section). Whenever a writer wants to enter the critical section the lock will ensure that all previous readers have finished before allowing the writer to enter the protected code. If a new reader or another writer wants now access to the critical section the lock prevents them to proceed until the current writer has finished.)

- organizing parallel processes / threads
  - master / slave (boss / worker)
    - can be used to distribute work to processes in distributed memory
    - slaves (workers) send a message to the master (boss) to request new work
    - slaves receive a message with their new work or with something like “all done”
    - slaves send a message to the master with their results, perhaps implicitly requesting more work
    - with some effort and a lot of synchronization this model can be used for threads in shared memory systems
  - producer / consumer (work pile, work queue)
    - similar to the *master / slave model* for shared memory
    - uses a shared queue to distribute work
    - producer threads create data chunks and store them into a shared queue
    - consumer threads take work from the queue whenever they need more work
    - possibly, consumer threads store their results into a result queue
      (One original producer or a special thread can handle the results in this case.)
    - access to the queue must be synchronized
    - this model is not suited for distributed memory

- granularity
  - often defined as the size of computation between communication or synchronization points
  - in distributed memory it must be guaranteed that the communication time does not dominate the overall execution time
  - the amount of computation must be large enough to outperform the overhead for communication and synchronization
  - coarse granularity (large number of sequential instructions) reduces the proportional costs of process / thread creation, synchronization, and communication and is often advantageous
  - fine granularity (very few sequential instructions) increases the number of concurrent processes / threads and helps to reduce overhead due to load imbalances
  - assume that the time for overhead per task is always the same

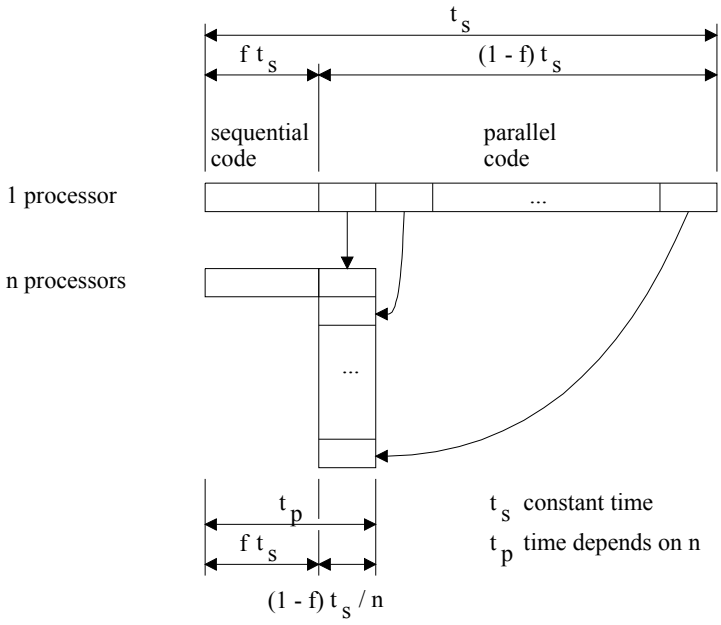


fine-grained decomposition coarse-grained decomposition

(In this artificial scenario, the fine-grained solution takes more time to finish the work on a four core system than the coarse-grained version. The coarse-grained solution may also profit from cache reuse or better optimization. The fine-grained solution takes less time on an eight or even sixteen core system because the coarse-grained version can still only use four cores. You must adapt the granularity to the special environment if that is possible.)

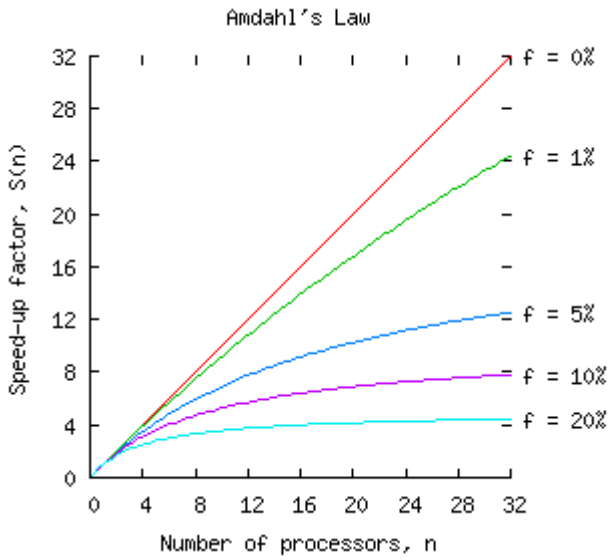
1.3 Speed-up and efficiency

- which speed-up is possible ?
  - maximum speed-up with *n* processors is *n* (linear speed-up)
    (in some rare cases a super-linear speed-up is possible due to cache reusability)
  - some limiting factors
    - some parts of a computation cannot be executed in parallel
    - some organizational tasks must be executed sequentially
      (initializing the system, creating processes / threads, etc.)
    - communication time for sending / receiving messages
    - time to synchronize the parallel computation
    - extra computations in the parallel version, e.g., recomputing values locally
  - assume that all serial parts are combined at the beginning and that all other parts can be executed in parallel with no overhead
    (*f* is the serial fraction, which cannot be executed in parallel)



– speed-up factor $S(n) = \frac{t_s}{t_p} = \frac{t_s}{ft_s + (1-f)t_s/n} = \frac{n}{1+(n-1)f}$

– equation is known as *Amdahl's law*



– speed-up is limited to $\lim_{n\to\infty} S(n) = \frac{1}{f}$

(with 5% serial and 95% parallel computation the maximum speed-up is 20)

– *Amdahl's law* represents the result for a problem with fixed size solved with an increasing number of processors to reduce the computation time

(Assume a painter wants to decorate a room with a new wallpaper and needs one hour to finish the work. It is possible that two painters can finish the work in half an hour. Is it possible that 60 painters finish the work in a minute or 3600 painters in a second? No, they would compete for resources (wallpaper, glue, table, etc.), get in the way of each other, or even do not have work to do because the room is too small for so many people. Perhaps it would even take more time than an hour to finish the work. This small example points to the problem of Amdahl's law who wants to solve a constant problem in shorter and shorter time. Assume on the other side that you have a hotel with 60 rooms or even some hotels with 3600 rooms in total, which should be new decorated. In this case you can finish all work in an hour if you have as many painters as rooms. This is the perspective of Gustafson who increases the amount of work when more workers are available.)

– *Gustafson* had stated that a larger multiprocessor system usually allows a larger size of the problem to be solved

  ♦ the size of the problem will be scaled so that the parallel execution time is fixed

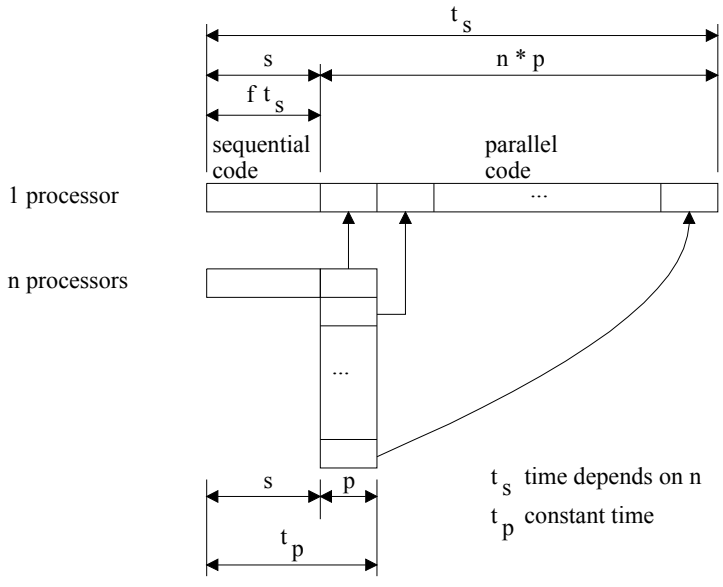
  ♦ the serial part of the code will generally not increase in the same way as the problem size

– *Gustafson's* scaled speed-up factor $S_s(n)$

  ♦ let s be the time to execute the serial part and p the time for executing the parallel part of the computation

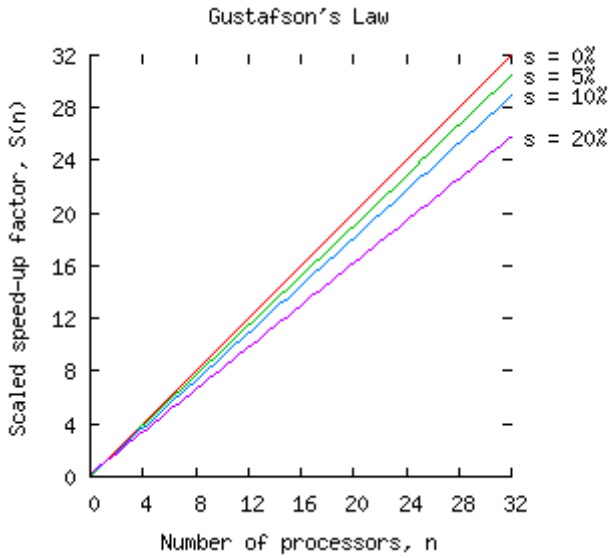
    ⇒ execution time on a parallel computer with n processors: $s + p$
    (the total execution time is fixed; for algebraic convenience: $s + p = 1$)

    ⇒ execution time on a serial computer: $s + n\,p$
    (all parallel parts must be executed sequentially)



$$S_s(n) = \frac{t_s}{t_p} = \frac{s + np}{s + p} \underset{s+p=1}{\Longrightarrow} \frac{s + n(1-s)}{s + 1 - s} = n + (1-n)s$$

- scaled speed-up



(with *Amdahl's law* the speed-up factor for a parallel computer with 32 processors and 5% serial code is 12.55 and with *Gustafson's law* the scaled speed-up factor is 30.45)

- do *Amdahl's law* and *Gustafson's law* contradict each other?

  - no, they use different meanings of "serial"

    (*f* is the serial fraction corresponding to the total work while *s* corresponds only to one parallel part and will not change when the number of parallel parts increases.)

  - correlation between serial parts in Amdahl's and Gustafson's law

$$ft_s = s \Rightarrow f = \frac{s}{s + np} = \frac{s}{s + n(1 - s)}$$

- now we can use Amdahl's law to compute the speed-up

$$S(n) = \frac{n}{1+(n-1)f} = \frac{n}{1+(n-1)\dfrac{s}{s+n(1-s)}}$$

$$S(n) = \frac{n(s+(1-s)n)}{s+(1-s)n+(n-1)s}$$

$$S(n) = \frac{ns+(1-s)n^2}{s+n-sn+ns-s} = \frac{ns+(1-s)n^2}{n}$$

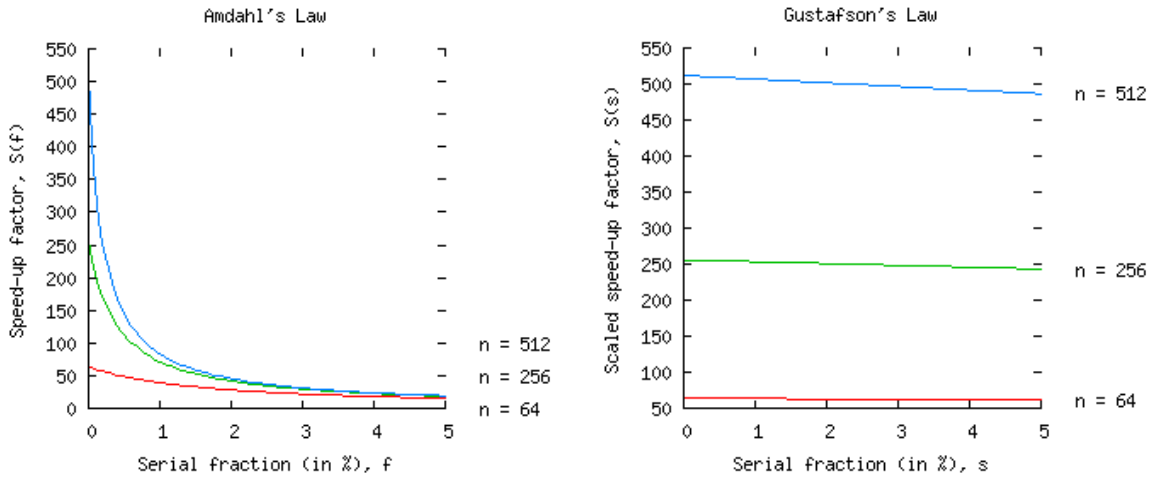
$$S(n) = s+(1-s)n = s+n-sn = n+(1-n)s$$

- $S(n)$ and $S_s(n)$ are equal if you use the same meaning for *serial part*

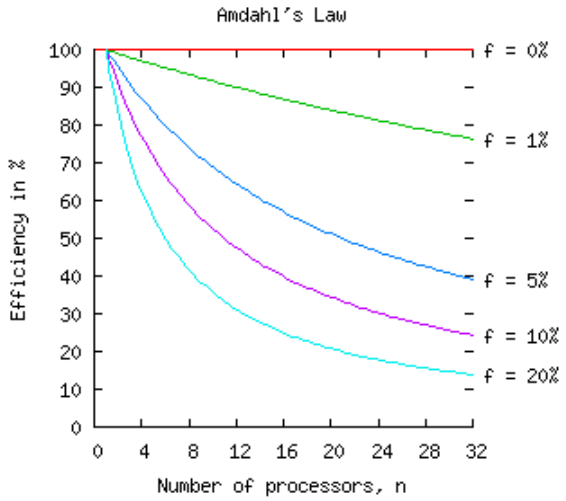
- the approach is different but the result is the same

  (The serial fraction of the total work will decrease if you increase the parallel part without increasing the sequential part. Amdahl's law is for problems which do not scale very well and which are not suited for a highly parallel computation, because you cannot scale the problem size. Gustafson's law is for problems which scale very good, so that both laws have their meaning.)

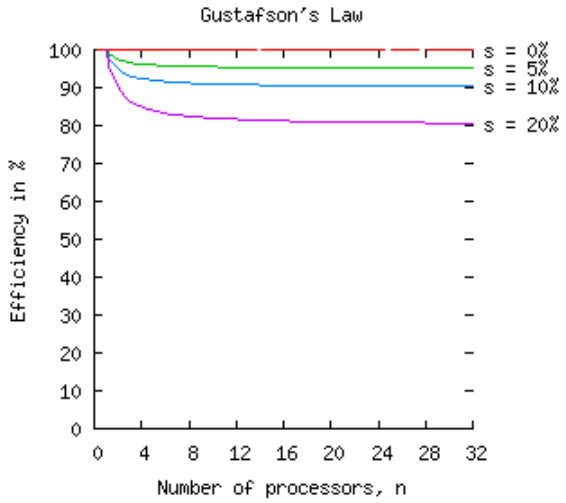
- speed-up as a function of the serial fraction



- speed-up is a metric to determine how much faster parallel execution is compared to serial execution
- efficiency shows how well the resources of the system are used
- $efficiency = \frac{speed\_up}{number\_of\_cores}\%$
- idle time of a processor or core = 100% - efficiency
- efficiency in accordance with Amdahl's law



- efficiency in accordance with Gustafson's law



1.4 Compiler techniques to parallelize programs

- this course focuses on how a programmer can write parallel programs
  - the programmer identifies parallel tasks
  - the programmer divides data among tasks
  - the programmer implements the necessary communication and synchronization mechanisms
- another approach is to use a parallelizing compiler
  - compiler takes a sequential program as input and converts it to a parallel program
  - compiler must identify independent tasks to do this work
  - compiler will also optimize the program
- how can you improve the performance of your program?
  - depends on the problem and on what is already available
    (you already have a program or you design and implement a new program from scratch)
  - let the compiler do the parallelization and all possible optimizations to increase the performance of your program
  - implement faster algorithms, improve your data structures if that is possible, provide hints where and how the compiler could automatically parallelize the program
  - develop a parallel program if you want to use a *cluster* (distributed memory) or if you are not satisfied with the optimized and automatically parallelized program

- dependence analysis
  - which parts of the program are independent
    $\Rightarrow$ candidates for parallel execution
  - which parts depend on each other
    $\Rightarrow$ require sequential execution or some form of synchronization
- compiler concentrates on possibilities to parallelize (nested) loops because in general a program spends most of its time in loops
- data dependence exists between two statements if they read or write a shared memory location in such a way that their execution order must be preserved to get always the same result
  - *true dependence* or *read-after-write dependence*

    a = b + c;
    d = a / 3;

    Second instruction reads the result of the first instruction.
  - *anti-dependence* or *write-after-read dependence*

    a = b + c;
    b = d / 3;

    Second instruction overwrites a value, which the first instruction reads.
  - *output dependence* or *write-after-write dependence*

    a = b + c;
    e = a * 3;
    a = d / 3;

    Second instruction reads the result of the first instruction (true dependence). First and third instructions write to the same memory location (output dependence).

- *input dependence* or *read-after-read dependence*

  a = b * 3;
  c = b / 3;

  Both instructions read the same memory location. This dependence does not restrict the execution order of the statements so that they can be executed in parallel.

- *independent* instructions

  a = b * 3;
  c = d / 3;

  Both instructions read and write different memory locations. Independent instructions can always be executed in parallel.

- it is more or less easy to determine a data dependence for scalar variables

- it is much harder or even impossible to determine a data dependence for loops, array references, or pointers

  - loop without dependence

```
for (i = 0; i < 100; ++i)
{
  a[i] = i;
}
```

  - loop with dependence

```
for (i = 0; i < 100; ++i)
{
  sum = sum + a[i];
}
```

  - loop with or without dependence

```
for (i = 0; i < 100; ++i)
{
  a[b[i]] = i;
}
```

    It depends on the values of array *b* if two or more iterations write to the same element of array *a*. There is no data dependence if *b* does not contain duplicate elements. The compiler does not know anything about the values of *b* so that it assumes dependence and does not parallelize the loop.

- compiler always assumes a dependence if it cannot guarantee an independence

- parallelizing a simple loop without data dependence

```
for (i = 0; i < n; ++i)
{
  a[i] = some complicated computation;
}
```

  - create one process / thread to compute *a[i]* for all even *i* and another one for all odd *i*

  - create some processes / threads, each one will compute an arbitrary but distinct block of the iteration space

  - create *n* processes / threads, each computing one *a[i]*

- some transformations to convert loops

  - strip mining (divides iteration space of one loop into two nested loops, row or column blocks)
  - loop blocking (row and column blocks, improves memory locality)
  - loop fission (loop distribution, splits the body of a loop into multiple separate loops where each body is a different subset of the original loop body, decreases granularity)
  - loop fusion (loop combining, reverse of loop fission, increases granularity)
  - loop splitting (breaks a loop into multiple loops which have the same bodies but iterate over different contiguous portions of the index space)
  - loop interchange (swaps inner and outer loops to rearrange the iteration space)
  - loop unrolling (unroll-and-jam, replicates loop body and does fewer iterations, increases granularity)
  - loop collapsing (combines nested loops into a single loop, reduces loop overhead)
  - pattern matching (replaces recognized patterns with calls to library functions)
  - privatization (gives each thread a copy of a variable)
  - scalar expansion (replaces many private variables by one array)

- **example 1:** Multiplication of two *n x n* matrices *a* and *b* storing the result into matrix *c*

  sequential code:

```
for (i = 0; i < n; ++i)
{
  for (j = 0; j < n; ++j)
  {
    sum = 0.0;
    for (k = 0; k < n; ++k)
    {
      sum = sum + a[i, k] * b[k, j];
    }
    c[i, j] = sum;
  }
}
```

- **example 2:** Given two *n x n* matrices *a* and *b* we want to store the transpose of matrix *a* into matrix *b*

  sequential code:

```
for (i = 0; i < n; ++i)
{
  for (j = 0; j < n; ++j)
  {
    b[i, j] = a[j, i];
  }
}
```

- **example 3:** Initializing two *n x n* matrices *a* and *b*

  sequential code:

```
for (i = 0; i < n; ++i)
{
  for (j = 0; j < n; ++j)
  {
    a[i, j] = i + j;
    b[i, j] = (i + j) / n;
  }
}
```

- strip mining
  - divides iterations of one loop into two nested loops
  - the outer loop iterates once per stripe and the inner loop iterates over a stripe
  - apply this transformation to example 1

    (assume *n* is a multiple of the stripe size *s*)

```
for (i = 0; i < n; i += s)         /* "n / s" stripes    */
{
  for (ii = i; ii < i + s; ++ii)   /* iter within stripe */
  {
    for (j = 0; j < n; ++j)
    {
      sum = 0.0;
      for (k = 0; k < n; ++k)
      {
        sum = sum + a[ii, k] * b[k, j];
      }
      c[ii, j] = sum;
    }
  }
}
```

  - the matrix multiplication works on row blocks if the outer loop will be parallelized

- loop blocking (loop tiling)
  - divides loop iteration space into smaller blocks / chunks
  - implicates a partitioning of a large array into smaller blocks so that accessed array elements fit into cache size
  - improves memory locality and cache reuse
  - apply this transformation to example 2

    C stores matrices in row-major order so that *b[i, .]* has a stride of 1 while *a[. ,i]* has a stride of *n* (length of row). If we read a value of *a* into a cache line, we also read subsequent values of the same row into the cache which we are not using yet in the sequential algorithm (and perhaps will never use if the cache line is invalid before we need the next value of this row). Let us assume that a cache line can store *S* elements of *a* or *b*. In this case we can use a *S x S* block to improve the sequential algorithm.

```
for (i = 0; i < n; i += S)
{
  for (j = 0; j < n; j += S)
  {
    for (ii = i; ii < MIN (i + S, n); ++ii)
    {
      for (jj = j; jj < MIN (j + S, n); ++jj)
      {
        b[ii, jj] = a[jj, ii];
      }
    }
  }
}
```

  - the innermost loops now access a square of *S x S* elements of matrix *a* and matrix *b* at a time so that they take full advantage of the cache
  - now we can additionally parallelize the outer loops so that different processes or threads work on different squares

- loop fission (loop distribution)
  - splits the body of one loop into multiple separate loops

    (Builds blocks of dependent statements where statements in different blocks are independent of each other. Every block ends up in a separate loop.)
  - apply this transformation to example 3

```
for (i = 0; i < n; ++i)          /* initialize matrix a */
{
  for (j = 0; j < n; ++j)
  {
    a[i, j] = i + j;
  }
}

for (i = 0; i < n; ++i)          /* initialize matrix b */
{
  for (j = 0; j < n; ++j)
  {
    b[i, j] = (i + j) / n;
  }
}
```

  - now we can use one process or thread to work on *a* and another one to work on *b*
  - loop distribution increases the number of processes or threads and decreases the granularity (less work per process / thread)
  - we can even parallelize each outer loop

- loop fusion (loop combining)
  - is the reverse of *loop fission*
  - combines two loops with the same header and independent bodies into one loop
  - reduces loop overhead
  - decreases the number of processes or threads and increases the granularity (more work per process / thread)

- loop splitting (loop peeling)
  - attempts to simplify a loop by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range
  - *loop peeling* is a special case of loop splitting
    - ♦ splits any problematic first or last few iterations from the loop (loop prolog, loop epilog)
    - ♦ performs them outside of the loop body
  - can for example be used if the „stripe size“ or „block size“ in the above examples is not a factor of the number of iterations

- loop interchange
  - swaps inner and outer loops to rearrange the iteration space
  - possible when the loops are independent of each other
  - can increase locality and therefore cache performance
  - example 3 assumes that matrices are stored in row-major order
  - it would be more efficient to update columns if they are stored in column-major order (like in Fortran)

    (The compiler can recognize if the programmer has used the “wrong” model and can improve cache performance through *loop interchange*.)

```
for (j = 0; j < n; ++j)
{
  for (i = 0; i < n; ++i)
  {
    a[i, j] = i + j;
    b[i, j] = (i + j) / n;
  }
}
```

- loop unrolling (unroll-and-jam)
  - replicates loop body and does fewer iterations

    (The step size of the loop will be changed from “1” to a “s” so that “s” iterations are unrolled to “1” iteration. The statements of the loop body will be replicated/jammed so that they are available for every leapfrogged iteration.)
  - optimizes execution speed at the expense of code size
  - reduces or even eliminates instructions that control the loop, e. g., incrementing loop variable or *end-of-loop* tests

    (For a small iteration space the loop can be completely eliminated. In that case this technique is sometimes called *loop unwinding*.)
  - apply this transformation to example 2 (assume *n* is a multiple of 4)

```
for (i = 0; i < n; i += 4)
{
  for (j = 0; j < n; ++j)
  {
    b[i, j]     = a[j, i];
    b[i + 1, j] = a[j, i + 1];
    b[i + 2, j] = a[j, i + 2];
    b[i + 3, j] = a[j, i + 3];
  }
}
```

  - increases the granularity of work if the outer loop will be parallelized

- loop collapsing
  - combines nested loops into a single loop to reduce loop overhead and to improve run-time performance
  - original code fragment

```
int a[P][Q];

for (i = 0; i < P; ++i)
{
  for (j = 0; j < Q; ++j)
  {
    a[i, j] = 0;
  }
}
```

  - code fragment after loop collapsing

```
int a[P][Q],
    *ptr_a = &a[0][0];

for (i = 0; i < P * Q; ++i)
{
  *ptr_a++ = 0;
}
```

  - can improve the chances for other optimizations, e.g., loop unrolling

- pattern matching
  - compiler recognizes common patterns for library functions, e.g., for *matrix-matrix* or *matrix-vector multiplications*
  - replaces recognized patterns with calls to library functions

    (often an optimized (commercial) library function provides a higher performance)
  - not always profitable

    (compiler attempts to determine if the cumulative time saved by using the optimized library routine will outperform the call overhead)
  - original code fragment

```
for (i = 0; i < MY_P; ++i)
{
  for (j = 0; j < Q; ++j)
  {
    c[i][j] = 0.0;
    for (k = 0; k < R; ++k)
    {
      c[i][j] += a[i, k] * b[k, j];
    }
  }
}
```

  - code fragment after pattern matching

```
double *const A = a[0],
       *const B = b[0],
       *const C = c[0];

dgemm ('N', 'N', MY_P, R, Q,
       1.0, A, MY_P,
            B, Q,
       0.0, C, MY_P);
```

- privatization
  - gives each thread a copy of a variable
  - apply this transformation to example 1

    (The compiler can parallelize the outer loop or even both outer loops if it *privatizes* variable *sum* for each process / thread and adds all partial sums into the original variable *sum* (*sum* is then a so-called *reduction variable*))

```
for (i = 0; i < n; ++i)
{
  for (j = 0; j < n; ++j)
  {
    sum = 0.0;
    for (k = 0; k < n; ++k)
    {
      sum = sum + a[i, k] * b[k, j];
    }
    c[i, j] = sum;
  }
}
```

- scalar expansion
  - replaces a scalar by an array
  - apply this transformation to example 1

    (The compiler can parallelize the outer loop or even both outer loops if it replaces variable *sum* by *c[i, j]*)

```
for (i = 0; i < n; ++i)
{
  for (j = 0; j < n; ++j)
  {
    c[i, j] = 0.0;
    for (k = 0; k < n; ++k)
    {
      c[i, j] = c[i, j] + a[i, k] * b[k, j];
    }
  }
}
```

  - this transformation removes also the assignment “c[i, j] = sum;”
  - “privatization” may be even better in this case because “scalar expansion” may result in “false sharing” of a cache line

- examples
  - line numbers of loops from *mat_mult_ijk.c*

```
59    for (i = 0; i < P; ++i)
60    {
61      for (j = 0; j < Q; ++j)
62      {
63        a[i][j] = 2.0;
64      }
65    }
...
68    for (i = 0; i < Q; ++i)
69    {
70      for (j = 0; j < R; ++j)
71      {
72        b[i][j] = 3.0;
73      }
74    }
...
79    for (i = 0; i < P; ++i)
80    {
81      for (j = 0; j < R; ++j)
82      {
83        c[i][j] = 0.0;
84        for (k = 0; k < Q; ++k)
85        {
86          c[i][j] += a[i][k] * b[k][j];
87        }
88      }
89    }
...
96    for (i = 0; i < P; ++i)
97    {
98      for (j = 0; j < R; ++j)
99      {
100       if (fabs (c[i][j] - tmp) > EPS)
101       {
102         ok++;
103       }
104     }
105   }
```

  - GNU gcc 7.2.0

loki introduction 109 **gcc -fopt-info -ftree-vectorize -ftree-parallelize-loops=12 -O2 mat_mult_ijk.c**
mat_mult_ijk.c:79:3: note: parallelizing outer loop 3
mat_mult_ijk.c:68:3: note: parallelizing outer loop 2
mat_mult_ijk.c:59:3: note: parallelizing outer loop 1
mat_mult_ijk.c:98:5: note: loop vectorized
mat_mult_ijk.c:94:7: note: basic block vectorized
mat_mult_ijk.c:70:5: note: loop vectorized
mat_mult_ijk.c:61:5: note: loop vectorized
loki introduction 110

loki introduction 110 **a.out**
```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                        elapsed time      cpu time
Mult "a" and "b":            7.00 s       91.80 s
```

loki introduction 111 **gcc -fopt-info -ftree-vectorize -O2 mat_mult_ijk.c**
mat_mult_ijk.c:98:5: note: loop vectorized
mat_mult_ijk.c:81:5: note: loop vectorized
mat_mult_ijk.c:70:5: note: loop vectorized
mat_mult_ijk.c:61:5: note: loop vectorized
loki introduction 112

loki introduction 112 **a.out**
```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                        elapsed time      cpu time
Mult "a" and "b":           23.00 s       23.03 s
```

– Intel Parallel Studio XE 2018 beta

loki introduction 113 **icc -qopt-matmul -qopt-report=1 mat_mult_ijk.c**
icc: remark #10397: optimization reports are generated in *.optrpt files in the output location

loki introduction 114 **more mat_mult_ijk.c.optrpt**
…
INLINE REPORT: (main(void))
  Report from: Loop nest, Vector & Auto-parallelization optimizations [loop, vec, par]
LOOP BEGIN at mat_mult_ijk.c(59,3)
  remark #25045: **Fused Loops: (59 68)**
  LOOP BEGIN at mat_mult_ijk.c(61,5)
    remark #25045: **Fused Loops: (61 70)**
    remark #15300: **LOOP WAS VECTORIZED**
  LOOP END
  LOOP BEGIN at mat_mult_ijk.c(**70**,5)
    remark #25046: **Loop lost in Fusion**
  LOOP END
LOOP END
LOOP BEGIN at mat_mult_ijk.c(**68**,3)
  remark #25046: **Loop lost in Fusion**
LOOP END
LOOP BEGIN at mat_mult_ijk.c(**79**,3)
<Distributed chunk1>
  remark #25426: **Loop Distributed (2 way)**
  LOOP BEGIN at mat_mult_ijk.c(**81**,5)
  <Distributed chunk1>
    remark #25426: **Loop Distributed (2 way)**
    remark #25408: **memset generated**
  LOOP END
LOOP END
LOOP BEGIN at mat_mult_ijk.c(**79**,3)
<Distributed chunk2>
  remark #25444: **Loopnest Interchanged**: (1 2 3) --> (1 3 2)
  remark #25459: **Loopnest replaced by matmul intrinsic**
LOOP END
LOOP BEGIN at mat_mult_ijk.c(**96**,3)
  remark #25460: No loop optimizations reported
  LOOP BEGIN at mat_mult_ijk.c(**98**,5)
    remark #15300: **LOOP WAS VECTORIZED**
  LOOP END
LOOP END
…

loki introduction 115 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                        elapsed time      cpu time
Mult "a" and "b":           0.00 s        1.03 s
```

– LLVM clang 5.0

loki introduction 116 **clang -O3 -Rpass-analysis=\.* -mllvm -polly -mllvm -polly-vectorizer=stripmine mat_mult_ijk.c**

```
mat_mult_ijk.c:52:9: remark: SCoP begins here. [-Rpass-analysis=polly-scops]
  int    i, j, k,                /* loop variables            */
         ^
mat_mult_ijk.c:86:23: remark: SCoP ends here. [-Rpass-analysis=polly-scops]
       c[i][j] += a[i][k] * b[k][j];
                       ^
mat_mult_ijk.c:50:5: remark: loop not vectorized: call instruction cannot be vectorized
     [-Rpass-analysis=loop-vectorize]
int main (void)
    ^
mat_mult_ijk.c:51:1: remark: 3342584 stack bytes in function [-Rpass-analysis=prologepilog]
{
^
mat_mult_ijk.c:51:1: remark: 918 instructions in function [-Rpass-analysis=asm-printer]
```

loki introduction 117 **clang -O3 -Rpass-missed=\.* -mllvm -polly -mllvm -polly-vectorizer=stripmine mat_mult_ijk.c**

```
mat_mult_ijk.c:86:10: remark: failed to move load with loop-invariant address because the
     loop may invalidate its value [-Rpass-missed=licm]
       c[i][j] += a[i][k] * b[k][j];
              ^
…
```

loki introduction 118 **clang -O3 -Rpass=\.* -mllvm -polly -mllvm -polly-vectorizer=stripmine mat_mult_ijk.c**

```
…
mat_mult_ijk.c:61:5: remark: vectorized loop (vectorization width: 2, interleaved count: 2)
     [-Rpass=loop-vectorize]
    for (j = 0; j < Q; ++j)
    ^
mat_mult_ijk.c:70:5: remark: vectorized loop (vectorization width: 2, interleaved count: 2)
     [-Rpass=loop-vectorize]
    for (j = 0; j < R; ++j)
    ^
mat_mult_ijk.c:98:5: remark: vectorized loop (vectorization width: 2, interleaved count: 2)
     [-Rpass=loop-vectorize]
    for (j = 0; j < R; ++j)
    ^
…
mat_mult_ijk.c:98:5: remark: unrolled loop by a factor of 2 with a breakout at trip 0
     [-Rpass=loop-unroll]
    for (j = 0; j < R; ++j)
    ^
mat_mult_ijk.c:50:5: remark: completely unrolled loop with 32 iterations
     [-Rpass=loop-unroll]
int main (void)
    ^
```

```
mat_mult_ijk.c:70:5: remark: unrolled loop by a factor of 4 with a breakout at trip 0
    [-Rpass=loop-unroll]
  for (j = 0; j < R; ++j)
  ^
mat_mult_ijk.c:61:5: remark: unrolled loop by a factor of 4 with a breakout at trip 0
    [-Rpass=loop-unroll]
  for (j = 0; j < Q; ++j)
  ^
…
```

loki introduction 119 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                        elapsed time      cpu time
Mult "a" and "b":           2.00 s        2.05 s
```

– Microsoft Visual Studio 2015

D:\...\introduction>**cl /GL /Gw /Ox /Qpar /Qpar-report:2 /Qvec-report:2 mat_mult_ijk.c**

Microsoft (R) C/C++-Optimierungscompiler Version 19.00.24215.1 für x64
Copyright (C) Microsoft Corporation. Alle Rechte vorbehalten.

mat_mult_ijk.c
Microsoft (R) Incremental Linker Version 14.00.24215.1
Copyright (C) Microsoft Corporation. All rights reserved.

/out:mat_mult_ijk.exe
/ltcg
mat_mult_ijk.obj
Code wird generiert.

--- Funktion wird analysiert: main
d:\...\introduction\mat_mult_ijk.c(61) : info C5002: Schleife aufgrund von "1300" nicht vektorisiert
d:\...\introduction\mat_mult_ijk.c(59) : info C5002: Schleife aufgrund von "1106" nicht vektorisiert
d:\...\introduction\mat_mult_ijk.c(70) : info C5002: Schleife aufgrund von "1300" nicht vektorisiert
d:\...\introduction\mat_mult_ijk.c(68) : info C5002: Schleife aufgrund von "1106" nicht vektorisiert
d:\...\introduction\mat_mult_ijk.c(84) : info C5002: Schleife aufgrund von "1203" nicht vektorisiert
d:\...\introduction\mat_mult_ijk.c(81) : info C5002: Schleife aufgrund von "1106" nicht vektorisiert
d:\...\introduction\mat_mult_ijk.c(79) : info C5002: Schleife aufgrund von "1106" nicht vektorisiert
d:\...\introduction\mat_mult_ijk.c(98) : info C5002: Schleife aufgrund von "1100" nicht vektorisiert
d:\...\introduction\mat_mult_ijk.c(96) : info C5002: Schleife aufgrund von "1106" nicht vektorisiert
d:\...\introduction\mat_mult_ijk.c(61) : info C5012: Schleife aufgrund von "1008" nicht parallelisiert
d:\...\introduction\mat_mult_ijk.c(59) : info C5012: Schleife aufgrund von "1008" nicht parallelisiert
d:\...\introduction\mat_mult_ijk.c(70) : info C5012: Schleife aufgrund von "1008" nicht parallelisiert
d:\...\introduction\mat_mult_ijk.c(68) : info C5012: Schleife aufgrund von "1008" nicht parallelisiert
d:\...\introduction\mat_mult_ijk.c(84) : info C5012: Schleife aufgrund von "1004" nicht parallelisiert
d:\...\introduction\mat_mult_ijk.c(81) : info C5012: Schleife aufgrund von "1008" nicht parallelisiert
d:\...\introduction\mat_mult_ijk.c(79) : info C5012: Schleife aufgrund von "1008" nicht parallelisiert
d:\...\introduction\mat_mult_ijk.c(98) : info C5012: Schleife aufgrund von "1008" nicht parallelisiert
d:\...\introduction\mat_mult_ijk.c(96) : info C5012: Schleife aufgrund von "1004" nicht parallelisiert
Codegenerierung ist abgeschlossen.

D:\...\introduction>**mat_mult_ijk.exe**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                        elapsed time      cpu time
Mult "a" and "b":          85.00 s       85.91 s
```

– Oracle Developer Studio 12.6

loki introduction 120 **cc -xautopar -xloopinfo mat_mult_ijk.c**
"mat_mult_ijk.c", line 59: PARALLELIZED, fused
"mat_mult_ijk.c", line 79: PARALLELIZED
"mat_mult_ijk.c", line 81: not parallelized, not profitable
"mat_mult_ijk.c", line 84: not parallelized, unsafe dependence (c)
"mat_mult_ijk.c", line 96: not parallelized, unsafe dependence (ok)
loki introduction 121

loki introduction 121 **a.out**
```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          8.00 s        8.18 s
```

– Portland Group compiler (Community Edition 2016)

loki introduction 122 **pgcc -Minfo=all mat_mult_ijk.c**

main:
61, Memory set idiom, **loop replaced by call to __c_mset8**
70, Memory set idiom, **loop replaced by call to __c_mset8**
81, **Loop distributed**: 2 new loops
Loop **interchange** produces reordered loop nest: **84,81**
Generated vector sse code for the loop
Generated 2 prefetch instructions for the loop
98, Loop not vectorized/parallelized: contains call
loki introduction 122

loki introduction 122 **a.out**
```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        5.25 s
```

Exercise 1-7:

Compile exercises 1-1 to 1-6 with automatic parallelization on *Linux* and *Windows* with *GNU C*, *Intel C*, *Microsoft C*, *Oracle Sun C*, and/or *Portland Group C*. Store all results with and without parallelization in a file.

1.5 Simple comparison of programming APIs

- at the moment no detailed explanation of the features
- just a visualization of the different APIs

1.5.1 Sequential dot product (dot_prod_sequential.c)

```
#define VECTOR_SIZE 100000000           /* vector size (10^8)           */
/* heap memory to avoid a segmentation fault due to a stack overflow    */
static double a[VECTOR_SIZE],           /* vectors for dot product      */
              b[VECTOR_SIZE];

int main (void)
{
  int i;                                /* loop variable                */
  double sum;
  /* initialize vectors                                                 */
  for (i = 0; i < VECTOR_SIZE; ++i)
  {
    a[i] = 2.0;
    b[i] = 3.0;
  }
  /* compute dot product                                              */
  sum = 0.0;
  for (i = 0; i < VECTOR_SIZE; ++i)
  {
    sum += a[i] * b[i];
  }
  printf ("sum = %e\n", sum);
  return EXIT_SUCCESS;
}
```

- ***sum*** is a so-called *reduction variable*, which accumulates the partial results of each iteration
- parallel programming APIs have often special functions or language constructs to support a reduction

1.5.2 Dot product with OpenMP (dot_prod_cyclic_OpenMP.c)

```
#ifdef _OPENMP
  #include <omp.h>
#endif

#define VECTOR_SIZE 100000000           /* vector size (10^8)           */

/* heap memory to avoid a segmentation fault due to a stack overflow    */
static double a[VECTOR_SIZE],           /* vectors for dot product      */
              b[VECTOR_SIZE];

int main (void)
{
  int i;                                /* loop variable                */
  double sum;

  /* initialize vectors                                                 */
  #pragma omp parallel for default(none) private(i) shared(a, b)
  for (i = 0; i < VECTOR_SIZE; ++i)
  {
    a[i] = 2.0;
    b[i] = 3.0;
  }

  /* compute dot product                                              */
  sum = 0.0;
  #pragma omp parallel for default(none) private(i) shared(a, b) \
    reduction(+:sum) schedule(static, 1)
  for (i = 0; i < VECTOR_SIZE; ++i)
  {
    #if defined _OPENMP && (VECTOR_SIZE < 20)
      printf ("Thread %d: i = %d.\n", omp_get_thread_num (), i);
    #endif
    sum += a[i] * b[i];
  }
  printf ("sum = %e\n", sum);
  return EXIT_SUCCESS;
}
```

- only a few extensions to the sequential program
- compiler needs a special option to take care of OpenMP pragmas

  (“gcc –fopenmp”, “cc –xopenmp”, “icc –openmp”, or “cl /openmp”. Compiler ignores OpenMP pragmas and creates a sequential program without this option.)
- environment variable OMP_NUM_THREADS determines the number of parallel threads, e.g., setenv OMP_NUM_THREADS 4
- the display/screen is a critical region, so that only one thread should write to it at a time (perhaps synchronization necessary)

  (the test output from all threads in the compute loop can intermingle and therefore be unreadable)

1.5.3 Dot product with Pthreads (dot_prod_cyclic_pthread.c)

```
#include <pthread.h>

#define VECTOR_SIZE 100000000          /* vector size (10^8)           */
#define NUM_THREADS 4                  /* default number of threads    */

/* Tests the return value of pthread functions.                        */
#define TestNotZero(val,file,line,function)  \
  if (val != 0) { fprintf (stderr, \
                  "File: %s, line %d: \"%s ()\" failed: %s\n", \
                  file, line, function, strerror (val)); \
                  exit (EXIT_FAILURE); }

/* global variables, so that all threads can easily access the values  */
static double a[VECTOR_SIZE],          /* vectors for dot product      */
              b[VECTOR_SIZE];
static int    nthreads;                /* number of threads            */

double *compute_dot_product (int *thr_num);

int main (int argc, char *argv[])
{
  /* Use "thr_num" to avoid a warning about a cast to pointer from
   * integer of different size with "gcc -Wint-to-pointer-cast -m64".
   * Every thread needs its own variable, so that we need an array.
   * "partial_sum" is an array of pointers to double, so that each
   * thread can return its pointer to its own partial sum.
   */
  int *thr_num,                        /* array of thread numbers      */
      ret;                             /* return value from functions  */
  double sum, **partial_sum;
  pthread_t *mytid;                    /* array of thread id's         */
  pthread_attr_t attr;                 /* thread attributes            */

  /* evaluate command line arguments                                   */
  switch (argc)
  {
    ...
  }

  /* allocate memory for all dynamic data structures                   */
  thr_num     = (int *) malloc ((size_t) nthreads * sizeof (int));
  mytid       =
   (pthread_t *) malloc ((size_t) nthreads * sizeof (pthread_t));
  partial_sum =
   (double **) malloc ((size_t) nthreads * sizeof (double *));
  if ((thr_num == NULL) || (mytid == NULL) || (partial_sum == NULL))
  {
    fprintf (stderr, "File: %s, line %d: Can't allocate memory.",
             __FILE__, __LINE__);
    exit (EXIT_FAILURE);
  }

  /* initialize vectors                                                */
  for (int i = 0; i < VECTOR_SIZE; ++i)
  {
    a[i] = 2.0;
    b[i] = 3.0;
  }

  /* initialize thread objects                                         */
  ret = pthread_attr_init (&attr);
  TestNotZero (ret, __FILE__, __LINE__, "pthread_attr_init");
  ret = pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_JOINABLE);
  TestNotZero (ret, __FILE__, __LINE__, "pthread_attr_setdetachstate");
```

```
  /* create threads                                                    */
  for (int i = 0; i < nthreads; ++i)
  {
    thr_num[i] = i;
    ret = pthread_create (&mytid[i], &attr,
                          (void * (*) (void *)) compute_dot_product,
                          (void *) &thr_num[i]);
    TestNotZero (ret, __FILE__, __LINE__, "pthread_create");
  }

  /* join threads and get result                                      */
  for (int i = 0; i < nthreads; ++i)
  {
    ret = pthread_join (mytid[i], (void **) &partial_sum[i]);
    TestNotZero (ret, __FILE__, __LINE__, "pthread_join");
  }

  /* compute and print sum                                            */
  sum = 0.0;
  for (int i = 0; i < nthreads; ++i)
  {
    sum += *partial_sum[i];
  }
  printf ("sum = %e\n", sum);

  /* clean up all things                                              */
  ret = pthread_attr_destroy (&attr);
  TestNotZero (ret, __FILE__, __LINE__, "pthread_attr_destroy");
  free (thr_num);
  free (mytid);
  for (int i = 0; i < nthreads; ++i)
  {
    free (partial_sum[i]);
  }
  free (partial_sum);
  return EXIT_SUCCESS;
}

double *compute_dot_product (int *thr_num)
{
  double *my_sum;

  my_sum = (double *) malloc (sizeof (double));
  if (my_sum == NULL)
  {
    ...
  }

  *my_sum = 0.0;
  for (int i = *thr_num; i < VECTOR_SIZE; i += nthreads)
  {
    #if VECTOR_SIZE < 20
      printf ("Thread %d: i = %d.\n", *thr_num, i);
    #endif
    *my_sum += a[i] * b[i];
  }
  pthread_exit ((void *) my_sum);
  /* The next statement isn't needed, but the compiler shouldn't
   * complain about a missing return-statement. The memory for
   * "my_sum" will be released at the end of function "main()".
   */
  return my_sum;
}
```

- *global variables* are necessary or you must set up a structure with all required values and pass it as parameter to each thread

- "main ()" must do the *reduction* of all partial sums of all threads

- all threads could update *sum* themselves if *sum* would be a global variable and the access to *sum* would be synchronized

  (would possibly reduce the performance due to the necessary synchronization)

- you must add "-lpthread" to the compiler command

- the display/screen is a critical region, so that only one thread should write to it at a time (perhaps synchronization necessary)

  (the test output from all threads in the compute loop can intermingle and therefore be unreadable)

1.5.4 Dot product with MPI (dot_prod_cyclic_MPI.c)

```
#include "mpi.h"

#define VECTOR_SIZE 100000000           /* vector size (10^8)            */

/* heap memory to avoid a segmentation fault due to a stack overflow    */
static double a[VECTOR_SIZE],           /* vectors for dot product       */
              b[VECTOR_SIZE];

int main (int argc, char *argv[])
{
  int mytid,                            /* task id (process rank)        */
      ntasks;                           /* number of processes           */
  double sum,
         my_sum;                        /* partial sum of one process    */

  MPI_Init (&argc, &argv);
  MPI_Comm_rank (MPI_COMM_WORLD, &mytid);
  MPI_Comm_size (MPI_COMM_WORLD, &ntasks);

  /* initialize vectors                                                 */
  for (int i = 0; i < VECTOR_SIZE; ++i)
  {
    a[i] = 2.0;
    b[i] = 3.0;
  }

  /* compute dot product                                                */
  my_sum = 0.0;
  for (int i = mytid; i < VECTOR_SIZE; i += ntasks)
  {
    #if VECTOR_SIZE < 20
      printf ("Process %d: i = %d.\n", mytid, i);
    #endif
    my_sum += a[i] * b[i];
  }
  MPI_Reduce (&my_sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (mytid == 0)
  {
    printf ("sum = %e\n", sum);
  }
  MPI_Finalize ();
  return EXIT_SUCCESS;
}
```

- you must compile with “mpicc”
- you must run the program with “mpiexec -np number_of_processes”
- the display/screen is a critical region, so that only the process with rank 0 should write to it

  (the test output from all processes in the compute loop can intermingle and therefore be unreadable)

1.5.5 Dot product with CUDA (dot_prod_CUDA.cu)

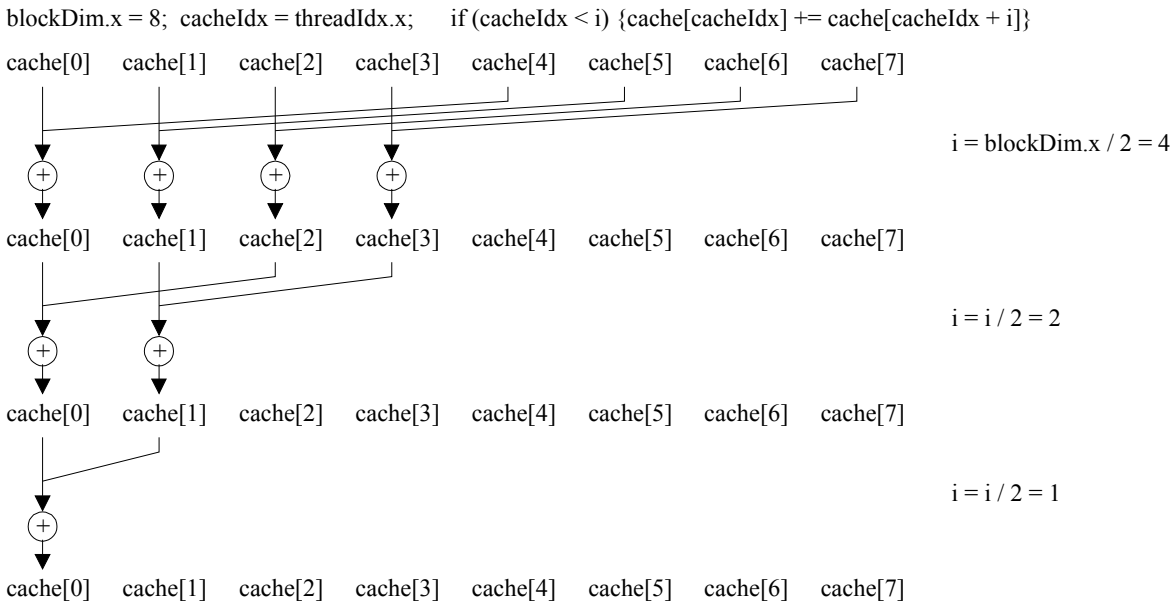
```
#define EPS               DBL_EPSILON   /* from float.h (2.2...e-16)    */

#define VECTOR_SIZE       10000000      /* vector size (10^8)           */
#define THREADS_PER_BLOCK 256           /* must be a power of two       */
#define BLOCKS_PER_GRID   32

/* Checks the return value of CUDA functions.                          */
#define CheckRetValueCudaFunction(val) \
  if (val != cudaSuccess) \
  { \
    fprintf (stderr, "file: %s  line %d: %s.\n", \
             __FILE__, __LINE__, cudaGetErrorString (val)); \
    cudaDeviceReset (); \
    exit (EXIT_FAILURE); \
  }

/* heap memory to avoid a segmentation fault due to a stack overflow   */
static double host_a[VECTOR_SIZE],      /* vectors for dot product     */
              host_b[VECTOR_SIZE];

__global__ void dot_prod (const double * __restrict__ a,
                          const double * __restrict__ b,
                          double * __restrict__ partial_sum);


int main (void)
{
  double sum,
         partial_sum[BLOCKS_PER_GRID],  /* result of each thread block  */
         *dev_a, *dev_b,                /* vector addresses on device   */
         *dev_partial_sum;
  cudaError_t ret;                      /* CUDA function return value   */

  /* assert that THREADS_PER_BLOCK is a power of two                   */
  assert (fabs (THREADS_PER_BLOCK - exp2 (log2 (THREADS_PER_BLOCK)))
          <= EPS);

  /* allocate memory for all vectors on the GPU (device)               */
  ret = cudaMalloc ((void **) &dev_a, VECTOR_SIZE * sizeof (double));
  CheckRetValueCudaFunction (ret);
  ret = cudaMalloc ((void **) &dev_b, VECTOR_SIZE * sizeof (double));
  CheckRetValueCudaFunction (ret);
  ret = cudaMalloc ((void **) &dev_partial_sum,
                    BLOCKS_PER_GRID * sizeof (double));
  CheckRetValueCudaFunction (ret);

  /* initialize vectors                                               */
  for (int i = 0; i < VECTOR_SIZE; ++i)
  {
    host_a[i] = 2.0;
    host_b[i] = 3.0;
  }

  /* copy vectors to the GPU                                          */
  ret = cudaMemcpy (dev_a, host_a, VECTOR_SIZE * sizeof (double),
                    cudaMemcpyHostToDevice);
  CheckRetValueCudaFunction (ret);
  ret = cudaMemcpy (dev_b, host_b, VECTOR_SIZE * sizeof (double),
                    cudaMemcpyHostToDevice);
  CheckRetValueCudaFunction (ret);

  /* compute partial dot products                                     */
  dot_prod <<<BLOCKS_PER_GRID, THREADS_PER_BLOCK>>>
    (dev_a, dev_b, dev_partial_sum);
```

```
  /* copy result vector "dev_partial_sum" back from the GPU to the CPU  */
  ret = cudaMemcpy (partial_sum, dev_partial_sum,
                    BLOCKS_PER_GRID * sizeof(double),
                    cudaMemcpyDeviceToHost);
  CheckRetValueCudaFunction (ret);

  /* compute "sum" of partial results                                   */
  sum = 0.0;
  for (int i = 0; i < BLOCKS_PER_GRID; ++i)
  {
    sum += partial_sum[i];
  }

  /* free allocated memory on the GPU                                   */
  ret = cudaFree (dev_a);
  CheckRetValueCudaFunction (ret);
  ret = cudaFree (dev_b);
  CheckRetValueCudaFunction (ret);
  ret = cudaFree (dev_partial_sum);
  CheckRetValueCudaFunction (ret);

  /* reset current device                                               */
  cudaDeviceReset ();

  printf ("sum = %e\n", sum);
  return EXIT_SUCCESS;
}

__global__ void dot_prod (const double * __restrict__ a,
                          const double * __restrict__ b,
                          double * __restrict__ partial_sum)
{
  /* Use shared memory to store each thread's running sum. The
   * compiler will allocate a copy of shared variables for each
   * block of threads   */
  __shared__ double cache[THREADS_PER_BLOCK];

  double temp = 0.0;
  int    cacheIdx = (int) threadIdx.x;

  for (int tid = (int) blockIdx.x * blockDim.x + threadIdx.x;
       tid < VECTOR_SIZE;
       tid += blockDim.x * gridDim.x)
  {
    temp += a[tid] * b[tid];
  }
  cache[cacheIdx] = temp;

  /* Ensure that all threads have completed, before you add up the
   * partial sums of each thread to the sum of the block   */
  __syncthreads ();

  /* Each thread will add two values and store the result back to
   * "cache". We need "log_2 (THREADS_PER_BLOCK)" steps to reduce
   * all partial values to one block value. THREADS_PER_BLOCK must
   * be a power of two for this reduction.   */
  for (int i = blockDim.x / 2; i > 0; i /= 2)
  {
    if (cacheIdx < i)
    {
      cache[cacheIdx] += cache[cacheIdx + i];
    }
    __syncthreads ();
  }
```

```
  /* store the partial sum of this thread block                    */
  if (cacheIdx == 0)
  {
    partial_sum[blockIdx.x] = cache[0];
  }
}
```

- you must compile with “nvcc”

- the parallel sum reduction tree



1.5.6 Dot product with OpenCL

- file with shared constants (dot_prod_OpenCL.h)

```
#define VECTOR_SIZE               10000000      /* vector size (10^7)   */
#define WORK_ITEMS_PER_WORK_GROUP 128           /* power of two required*/
#define WORK_GROUPS_PER_NDRANGE    32
```

- OpenCl *kernel* (dotProdOpenCL.cl)

```
#if defined (cl_khr_fp64) || defined (cl_amd_fp64)
  #include "dot_prod_OpenCL.h"

  __kernel void dotProdKernel (__global const double *restrict a,
                               __global const double *restrict b,
                               __global double *restrict partial_sum)
  {
    /* Use local memory to store each work-items running sum.          */
    __local double cache[WORK_ITEMS_PER_WORK_GROUP];

    double temp = 0.0;
    int    cacheIdx = get_local_id (0);

    for (int tid = get_global_id (0);
         tid < VECTOR_SIZE;
         tid += get_global_size (0))
    {
      temp += a[tid] * b[tid];
    }
    cache[cacheIdx] = temp;

    /* Ensure that all work-items have completed, before you add up the
     * partial sums of each work-item to the sum of the work-group
     */
    barrier (CLK_LOCAL_MEM_FENCE);

    /* Each work-item will add two values and store the result back to
     * "cache". We need "log_2 (WORK_ITEMS_PER_WORK_GROUP)" steps to
     * reduce all partial values to one work-group value.
     * WORK_ITEMS_PER_WORK_GROUP must be a power of two for this
     * reduction.
     */
    for (int i = get_local_size (0) / 2; i > 0; i /= 2)
    {
      if (cacheIdx < i)
      {
        cache[cacheIdx] += cache[cacheIdx + i];
      }
      barrier (CLK_LOCAL_MEM_FENCE);
    }
    /* store the partial sum of this thread block                      */
    if (cacheIdx == 0)
    {
      partial_sum[get_group_id (0)] = cache[0];
    }
  }
#else
  #error "Double precision floating point not supported."
#endif
```

- OpenCL *host*-program (dot_prod_OpenCL.c)

```
#define CL_USE_DEPRECATED_OPENCL_1_2_APIS
...
#ifndef __APPLE__
  #include <CL/cl.h>
#else
  #include <OpenCL/opencl.h>
#endif
#include "dot_prod_OpenCL.h"            /* include some constants       */

#define EPS         DBL_EPSILON         /* from float.h (2.2...e-16)    */

#define KERNEL_NAME "dotProdKernel"
#define FILENAME    "dotProdKernel.cl"
#define MAX_FNAME_LENGTH          256

#if (defined(WIN32) || defined(_WIN32) || defined(Win32)) && \
     !defined(Cygwin)
  #define PATH_SEPARATOR "\\"
#else
  #define PATH_SEPARATOR "/"
#endif

...
/* Define macro to check the return value of an OpenCL function. The
 * function prototype is necessary, because the compiler will assume
 * that "getErrorName ()" will return "int" without a prototype.
 */
const char *getErrorName (cl_int errCode);
#define CheckRetValueOfOpenCLFunction(val) \
  if (val != CL_SUCCESS) \
  { \
    fprintf (stderr, "file: %s  line %d: %s.\n", \
             __FILE__, __LINE__, getErrorName (val)); \
    exit (EXIT_FAILURE); \
  }

/* heap memory to avoid a segmentation fault due to a stack overflow    */
static double a[VECTOR_SIZE],           /* vectors for dot product      */
              b[VECTOR_SIZE];

int main (void)
{
  FILE          *fp;                    /* for kernelSource             */
  const char    *kernelSrc;             /* necessary to avoid warning   */
  char          *kernelSource,
                *kernelDirectory,
                fname[MAX_FNAME_LENGTH], /* filename incl. pathname     */
                *paramValue,
                *programBuildOptions;
  int           retVal,                 /* return value                 */
                kernelSize;             /* size of kernel code          */
  double        sum,                    /* result of dot product        */
                                        /* result of each work_group    */
                partial_sum[WORK_GROUPS_PER_NDRANGE];
  size_t        paramValueSize,
                localWorkSize, globalWorkSize;
  cl_int        errcode_ret,            /* returned error code          */
                ret;                    /* OpenCL function return value */
  cl_uint       numPlatforms,           /* # of available platforms     */
                numDevices = 0;         /* # of available devices       */
  cl_mem        dev_a, dev_b,
                dev_partial_sum;
```

```
cl_platform_id *platform_ids,         /* platform IDs                 */
               platform_id;
cl_device_id   device_id = NULL;      /* device ID                    */
cl_context     context;
cl_command_queue command_queue;
cl_program       program;
cl_kernel        kernel;

/* assert that WORK_ITEMS_PER_WORK_GROUP is a power of two           */
assert (fabs (WORK_ITEMS_PER_WORK_GROUP -
              exp2 (log2 (WORK_ITEMS_PER_WORK_GROUP))) <= EPS);

/*************************************************************
 *
 * Step 1: Get all platform IDs and check for a GPU device
 *
 *************************************************************
 */
/* get the number of available platforms                             */
numPlatforms = 0;
ret = clGetPlatformIDs (0, NULL, &numPlatforms);
if (numPlatforms > 0)
{
  printf ("\nFound %d platform(s).\n", numPlatforms);
}
else
{
  printf ("\nCouldn't find any OpenCL capable platforms.\n\n");
  return EXIT_SUCCESS;
}
/* get platform IDs                                                  */
platform_ids = (cl_platform_id *) malloc (numPlatforms *
                                          sizeof (cl_platform_id));
TestEqualsNULL (platform_ids);
ret = clGetPlatformIDs (numPlatforms, platform_ids, NULL);
CheckRetValueOfOpenCLFunction (ret);

/*************************************************************
 *
 * Step 2: Get a device ID
 *
 *************************************************************
 */
/* try to get a device ID for a GPU first                            */
printf ("Try to find first GPU on available platforms.\n");
for (unsigned int platform = 0; platform < numPlatforms; ++platform)
{
  printf ("  ********  Using platform %u  ********\n", platform);
  numDevices = 0;
  platform_id = platform_ids[platform];
  ret = clGetDeviceIDs (platform_id, CL_DEVICE_TYPE_GPU,
                        1, &device_id, &numDevices);
  /* Probably returns "CL_DEVICE_NOT_FOUND" so that it is not allowed
   * to use "CheckRetValueOfOpenCLFunction (ret);" The result must be
   * checked with "numDevices".
   */
  if (numDevices == 0)
  {
    printf ("    Couldn't find a GPU device on platform %u.\n"
            "    Try next platform.\n\n", platform);
    continue;                         /* try next platform           */
  }
  else
```

```
    {
      break;                            /* GPU is available            */
    }
  }
  /* try to get a device ID for a CPU next                              */
  if (numDevices == 0)
  {
    printf ("Try to find first CPU on available platforms.\n");
    for (unsigned int platform = 0; platform < numPlatforms; ++platform)
    {
      printf ("  ********  Using platform %u  ********\n", platform);
      platform_id = platform_ids[platform];
      ret = clGetDeviceIDs (platform_id, CL_DEVICE_TYPE_CPU,
                            1, &device_id, &numDevices);
      /* Probably returns "CL_DEVICE_NOT_FOUND" so that it is not
       * allowed to use "CheckRetValueOfOpenCLFunction (ret);" The
       * result must be checked with "numDevices".
       */
      if (numDevices == 0)
      {
        printf ("    Could neither find a CPU device on platform %u\n."
                "    Try next platform.\n\n", platform);
        continue;                       /* try next platform           */
      }
      else
      {
        break;                          /* CPU is available            */
      }
    }
  }
  free (platform_ids);
  if (numDevices > 0)
  {
    /* get device name                                                 */
    ret = clGetDeviceInfo (device_id, CL_DEVICE_NAME,
                           0, NULL, &paramValueSize);
    CheckRetValueOfOpenCLFunction (ret);
    paramValue =
      (char *) malloc (paramValueSize * sizeof (char));
    TestEqualsNULL (paramValue);
    ret = clGetDeviceInfo (device_id, CL_DEVICE_NAME,
                           paramValueSize, paramValue, NULL);
    CheckRetValueOfOpenCLFunction (ret);
    printf ("    Use device %s.\n\n", paramValue);
    fflush (stdout);
    free (paramValue);
  }
  else
  {
    printf ("\nCouldn't find any GPU or CPU.\n\n");
    return EXIT_SUCCESS;
  }

  /*************************************************************
   *
   * Step 3: Create context for device "device_id"
   *
   *************************************************************
   */
  context = clCreateContext (NULL, 1, &device_id,
                             NULL, NULL, &errcode_ret);
  CheckRetValueOfOpenCLFunction (errcode_ret);
```

```
/*************************************************************
 *
 * Step 4: Create kernel as a string
 *
 *************************************************************
 */
/* read kernel from file                                      */
memset (fname, 0, MAX_FNAME_LENGTH);
kernelDirectory = getenv ("KERNEL_FILES");
if (kernelDirectory != NULL)
{
  strncpy (fname, kernelDirectory, MAX_FNAME_LENGTH - 1);
  /* check, if the last character of the environment variable
   * KERNEL_FILES is a path separator and add one otherwise
   */
  if (fname[strlen (fname) - 1] != PATH_SEPARATOR[0])
  {
    strncat (fname, PATH_SEPARATOR,
             MAX_FNAME_LENGTH - strlen (fname) - 1);
  }
}
strncat (fname, FILENAME, MAX_FNAME_LENGTH - strlen (fname) - 1);
fp = fopen (fname, "r");
if (fp == NULL)
{
  fprintf (stderr, "file: %s  line %d: Couldn't open file "
           "\"%s\".\n", __FILE__, __LINE__, FILENAME);
  exit (EXIT_FAILURE);
}
retVal = fseek (fp, 0, SEEK_END);
if (retVal != 0)
{
  fprintf (stderr, "file: %s  line %d: \"fseek ()\" failed: "
           "\"%s\".\n", __FILE__, __LINE__, strerror (retVal));
  exit (EXIT_FAILURE);
}
kernelSize = (int) ftell (fp);
rewind (fp);
kernelSource = (char *) malloc ((size_t) (kernelSize + 1));
TestEqualsNULL (kernelSource);
/* make sure that the string is \0 terminated                 */
kernelSource[kernelSize] = '\0';
fread (kernelSource, sizeof (char), (size_t) kernelSize, fp);
if (ferror (fp) != 0)
{
  fprintf (stderr, "file: %s  line %d: \"fread ()\" failed.\n",
           __FILE__, __LINE__);
  exit (EXIT_FAILURE);
}
fclose (fp);

/*************************************************************
 *
 * Step 5: Create program object
 *
 *************************************************************
 */
/* Without "kernelSrc" "gcc -Wcast-qual" displays the following
 * warning "... warning: to be safe all intermediate pointers in
 * cast from 'char **' to 'const char **' must be 'const'
 * qualified [-Wcast-qual]".
 */
kernelSrc = kernelSource;
```

```
program = clCreateProgramWithSource (context, 1,
            (const char **) &kernelSrc, NULL, &errcode_ret);
CheckRetValueOfOpenCLFunction (errcode_ret);
kernelSrc = NULL;
free (kernelSource);

/*************************************************************
 *
 * Step 6: Build program executables
 *
 *************************************************************
 */
#ifdef CL_VERSION_2_0
  programBuildOptions = "-cl-std=CL2.0";
#else
  programBuildOptions = NULL;
#endif
ret = clBuildProgram (program, 1, &device_id,
                      programBuildOptions, NULL, NULL);
if (ret != CL_SUCCESS)
{
  /* check log file                                                */
  ret = clGetProgramBuildInfo (program, device_id,
                               CL_PROGRAM_BUILD_LOG, 0, NULL,
                               &paramValueSize);
  CheckRetValueOfOpenCLFunction (ret);
  if (paramValueSize > 0)
  {
    paramValue =
      (char *) malloc ((paramValueSize * sizeof (char)) + 1);
    TestEqualsNULL (paramValue);
    /* make sure that the string is \0 terminated                  */
    paramValue[paramValueSize] = '\0';
    ret = clGetProgramBuildInfo (program, device_id,
                                 CL_PROGRAM_BUILD_LOG,
                                 paramValueSize, paramValue, NULL);
    CheckRetValueOfOpenCLFunction (ret);
    printf ("\nCompiler log file:\n\n%s", paramValue);
    free (paramValue);
    exit (EXIT_FAILURE);
  }
}

/*************************************************************
 *
 * Step 7: Create buffer for kernel on device "device_id"
 *
 *************************************************************
 */
/* initialize vectors                                          */
for (int i = 0; i < VECTOR_SIZE; ++i)
{
  a[i] = 2.0;
  b[i] = 3.0;
}
dev_a = clCreateBuffer (context,
                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                        VECTOR_SIZE * sizeof (double), a,
                        &errcode_ret);
CheckRetValueOfOpenCLFunction (errcode_ret);
```

```
dev_b = clCreateBuffer (context,
                        CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                        VECTOR_SIZE * sizeof (double), b,
                        &errcode_ret);
CheckRetValueOfOpenCLFunction (errcode_ret);
dev_partial_sum = clCreateBuffer (context,
  CL_MEM_WRITE_ONLY, WORK_GROUPS_PER_NDRANGE * sizeof (double),
  NULL, &errcode_ret);
CheckRetValueOfOpenCLFunction (errcode_ret);

/*************************************************************
 *
 * Step 8: Create kernel object
 *
 *************************************************************
 */
kernel = clCreateKernel (program, KERNEL_NAME, &errcode_ret);
CheckRetValueOfOpenCLFunction (errcode_ret);

/*************************************************************
 *
 * Step 9: Set kernel arguments
 *
 *************************************************************
 */
ret = clSetKernelArg (kernel, 0, sizeof (cl_mem), (void *) &dev_a);
CheckRetValueOfOpenCLFunction (ret);
ret = clSetKernelArg (kernel, 1, sizeof (cl_mem), (void *) &dev_b);
CheckRetValueOfOpenCLFunction (ret);
ret = clSetKernelArg (kernel, 2, sizeof (cl_mem),
                      (void *) &dev_partial_sum);
CheckRetValueOfOpenCLFunction (ret);

/*************************************************************
 *
 * Step 10: Create command queue for device "device_id"
 *
 *************************************************************
 */
command_queue = clCreateCommandQueue (context, device_id, 0,
                                      &errcode_ret);
CheckRetValueOfOpenCLFunction (errcode_ret);

/*************************************************************
 *
 * Step 11: Enqueue a command to execute a kernel
 *
 *************************************************************
 */
localWorkSize  = WORK_ITEMS_PER_WORK_GROUP;
globalWorkSize = localWorkSize * WORK_GROUPS_PER_NDRANGE;
ret = clEnqueueNDRangeKernel (command_queue, kernel, 1, NULL,
                              &globalWorkSize, &localWorkSize,
                              0, NULL, NULL);
CheckRetValueOfOpenCLFunction (ret);

/* finalize                                                    */
ret = clFlush (command_queue);
CheckRetValueOfOpenCLFunction (ret);
ret = clFinish (command_queue);
CheckRetValueOfOpenCLFunction (ret);
```

```
  /*************************************************************
   *
   * Step 12: Read memory objects (result)
   *
   *************************************************************
   */
  ret = clEnqueueReadBuffer (command_queue, dev_partial_sum,
    CL_TRUE, 0, WORK_GROUPS_PER_NDRANGE * sizeof (double),
    partial_sum, 0, NULL, NULL);
  CheckRetValueOfOpenCLFunction (ret);
  /* compute "sum" of partial results                          */
  sum = 0.0;
  for (int i = 0; i < WORK_GROUPS_PER_NDRANGE; ++i)
  {
    sum += partial_sum[i];
  }
  printf ("sum = %e\n", sum);

  /*************************************************************
   *
   * Step 13: Free objects
   *
   *************************************************************
   */
  ret = clReleaseKernel (kernel);
  CheckRetValueOfOpenCLFunction (ret);
  ret = clReleaseProgram (program);
  CheckRetValueOfOpenCLFunction (ret);
  ret = clReleaseMemObject (dev_a);
  CheckRetValueOfOpenCLFunction (ret);
  ret = clReleaseMemObject (dev_b);
  CheckRetValueOfOpenCLFunction (ret);
  ret = clReleaseMemObject (dev_partial_sum);
  CheckRetValueOfOpenCLFunction (ret);
  ret = clReleaseCommandQueue (command_queue);
  CheckRetValueOfOpenCLFunction (ret);
  ret = clReleaseContext (context);
  CheckRetValueOfOpenCLFunction (ret);

  return EXIT_SUCCESS;
}
```

1.5.7 Dot product with OpenACC (dot_prod_OpenACC.c)

```
#ifdef _OPENACC
  #include <openacc.h>
#endif

#define VECTOR_SIZE 100000000           /* vector size (10^8)           */

/* heap memory to avoid a segmentation fault due to a stack overflow    */
static double a[VECTOR_SIZE],           /* vectors for dot product      */
              b[VECTOR_SIZE];

int main (void)
{
  double sum;

  /* initialize vectors                                                 */
  #pragma acc parallel loop independent \
    copyout(a[0:VECTOR_SIZE], b[0:VECTOR_SIZE])
  for (int i = 0; i < VECTOR_SIZE; ++i)
  {
    a[i] = 2.0;
    b[i] = 3.0;
  }

  #ifdef _OPENACC
    printf ("Supported standard:                _OPENACC = %d\n"
            "Number of host devices:            %d\n"
            "Number of none host devices:       %d\n"
            "Number of attached NVIDIA devices: %d\n",
            _OPENACC,
            acc_get_num_devices(acc_device_host),
            acc_get_num_devices(acc_device_not_host),
            acc_get_num_devices(acc_device_nvidia));
  #endif

  /* compute dot product                                                */
  sum = 0.0;
  #pragma acc data copyin(a[0:VECTOR_SIZE], b[0:VECTOR_SIZE]) copy(sum)
  #pragma acc parallel loop independent reduction(+:sum)
  for (int i = 0; i < VECTOR_SIZE; ++i)
  {
    sum += a[i] * b[i];
  }
  printf ("sum = %e\n", sum);
  return EXIT_SUCCESS;
}
```

- only a few extensions to the sequential program
- compiler needs special options to take care of OpenACC pragmas and to create a program for a device

  (“gcc -fopenacc [-foffload-target=nvptx-none]”, “pgcc -acc -ta=nvidia”. You can specify a target with “-foffload-target=”. “gcc” creates programs for all targets, which were specified at compile time of the compiler, without that option. Compiler ignores OpenACC pragmas and creates a sequential program without these options.)
- “gcc-5.x” or newer supports offloading to NVIDIA devices

1.5.8 Dot product with OpenMP for accelerators

(dot_prod_accelerator_OpenMP.c)

```
#ifdef _OPENMP
  #include <omp.h>
#endif

#define VECTOR_SIZE 100000000           /* vector size (10^8)           */

/* heap memory to avoid a segmentation fault due to a stack overflow    */
static double a[VECTOR_SIZE],           /* vectors for dot product      */
              b[VECTOR_SIZE];

int main (void)
{
  double sum;

  /* initialize vectors                                                 */
  #pragma omp target map (from: a, b)
  #pragma omp parallel for default(none) shared(a, b)
  for (int i = 0; i < VECTOR_SIZE; ++i)
  {
    a[i] = 2.0;
    b[i] = 3.0;
  }

  #ifdef _OPENMP
    printf ("Number of processors:     %d\n"
            "Number of devices:        %d\n" ...,
            omp_get_num_procs (), omp_get_num_devices (), ...);
  #endif

  /* compute dot product                                                */
  sum = 0.0;
  #pragma omp target map(to:a,b), map(tofrom:sum)
  #pragma omp parallel for default(none) shared(a, b) reduction(+:sum)
  for (int i = 0; i < VECTOR_SIZE; ++i)
  {
    sum += a[i] * b[i];
  }
  printf ("sum = %e\n", sum);
  return EXIT_SUCCESS;
}
```

- only a few extensions to the sequential program
- compiler needs special options to take care of OpenMP pragmas for accelerators and to create a program for a device

  (“gcc -fopenmp [-foffload-target=nvptx-none]”. You can specify a target with “-foffload-target=”. “gcc” will build offload images for all offload targets, which were specified at compile time of the compiler, without that option. Compiler ignores OpenMP pragmas and creates a sequential program without these options.)
- “gcc-7.x” or newer supports offloading to NVIDIA devices

Appendix A Compiler options

- some C compilers and their options to optimize and parallelize programs for *Microsoft Windows*, *Cygwin*, *Mac OS X*, *Linux*, and/or *Solaris* (SunOS)

- sometimes an option is a shortcut for a bundle of options

A.1 GNU gcc 7.x

- -falign-functions=n
  Aligns the start of functions to the next power-of-two greater than *n*, skipping up to *n* bytes. "-falign-functions=32" aligns functions to the next 32-byte boundary, but "-falign-functions=24" would align to the next 32-byte boundary only if this can be done by skipping 23 bytes or less. "-falign-functions=1" will not align any functions. This is the default if you compile with "-O2 or higher".

- -falign-loops=n
  Aligns loops to a power-of-two boundary, skipping up to n bytes like "-falign-functions". "-falign-loops=1" will not align any loops. This is the default if you compile with "-O2 or higher".

- -finline-functions
  Integrates all simple functions into their callers. The compiler heuristically decides which functions are simple enough for this operation. This is the default if you compile with "-O3".

- -finline-functions-called-once
  Considers all static functions called once for inlining into their caller even if they are not marked inline. This is the default if you compile with "-O1" or higher.

- -finline-small-functions
  Integrates functions into their callers when their body is smaller than the expected function call code. The compiler heuristically decides which functions are small enough for this operation. This is the default if you compile with "-O2".

- -floop-block
  Performs loop blocking transformation on loops. The block length can be changed using the parameter *loop-block-tile-size*.

- -floop-interchange
  Performs loop interchange transformation on loops. Interchanging two nested loops, switches the inner and outer loop.

- -floop-parallelize-all
  Parallelizes all loops that do not contain data dependence without checking that it is profitable to parallelize the loops.

- -floop-strip-mine
  Performs loop strip mining transformation on loops. Strip mining splits a loop into two nested loops. The outer loop has strides equal to the strip size and the inner loop has strides of the original loop within a strip.

- -foffload=[<targets>[=<options>] | <options>]
  By default, GCC will build offload images for all offload targets, which were specified at compile time of the compiler, with non-target-specific options passed to the host compiler. This option can be used to control offload targets and options for them, e. g.,

-foffload=nvptx-none	Build offload images for target “nvptx-none” with non-target-specific options passed to the host compiler.
-foffload=nvptx-none=”-O3”	Build offload images for target “nvptx-none” with non-target-specific options passed to the host compiler plus option “-O3”.
-foffload=”-O3”	Build offload images for all targets specified at compile time of the compiler with non-target-specific options passed to the host compiler plus option “-O3”.
-foffload=-lm	Link in the math library on accelerator targets.

  <targets> can be separated by commas and several <options> can be separated by spaces. Options specified by *-foffload* are appended to the end of the option set, so in case of option conflicts they have more priority. The *-foffload* flag can be specified several times, and you have to do that to specify different <options> for different <targets>.

- -fopenacc
  Enables explicit parallelization with OpenACC directives and defines the preprocessor token _OPENACC. When *-fopenacc* is specified, the compiler generates accelerated code according to the OpenACC Application Programming Interface v2.0. You can set the environment variables ACC_DEVICE_TYPE and ACC_DEVICE_NUM prior to the execution of an accelerated program to specify which device should be used, e. g.,

setenv ACC_DEVICE_TYPE host	(use a single-threaded process on a CPU),
setenv ACC_DEVICE_TYPE nvidia	(use a NVIDIA GPU).
setenv ACC_DEVICE_NUM *n*	(0 <= *n* < number of installed accelerator devices)

- -fopenacc-dim=<gang:worker:vector>
  Specifies default compute dimensions for parallel constructs when there are no explicit user defined clauses on the directive. The *gang*, *worker*, and *vector* argument must be a positive

integer or a minus sign. A minus sign allows the runtime to determine a suitable default based on the accelerator's capabilities. The default for NVIDIA accelerators is "-:32:32".

- -fopenmp
  Enables explicit parallelization with OpenMP directives and defines the preprocessor token _OPENMP. When *-fopenmp* is specified, the compiler generates parallel/accelerated code according to the OpenMP Application Program Interface v4.5. You can set the environment variable OMP_NUM_THREADS prior to the execution of a parallelized program to specify how many threads should be used and the environment variable OMP_NESTED to enable nested parallelism. You can set the environment variable OMP_DEFAULT_DEVICE prior to the execution of an accelerated program to specify which device should be used, e. g.,

  setenv OMP_DEFAULT_DEVICE 0 (use a CPU),
  setenv OMP_DEFAULT_DEVICE 1 (use a NVIDIA GPU).

- -fopenmp-simd
  Enable only handling of OpenMP's SIMD directives. Other OpenMP directives are ignored.

- -fopt-info[-options[=<filename>]]
  Emit useful information from various optimization passes. If "-options" is used, *options* is a list of "-" separated option keywords to select the details and if "=<filename>" is provided, the output is stored in "<filename>" (otherwise it is printed onto *stderr*). The *options* can be divided into two groups: options describing the verbosity of the dump, and options describing which optimizations should be included. The options from both groups can be freely mixed as they are non-overlapping. However, in case of any conflicts, the later options override earlier options on the command line. The following options control the dump verbosity:

optimized	Print information when an optimization is successfully applied.
missed	Print information about missed optimizations.
note	Print verbose information about optimizations, such as certain transformations, more detailed messages about decisions etc.
all	Print detailed optimization information. This includes *optimized*, *missed*, and *note*.

  One or more of the following option keywords can be used to describe a group of optimizations:

ipa	Enable dumps from all interprocedural optimizations.
loop	Enable dumps from all loop optimizations.
inline	Enable dumps from all inlining optimizations.
omp	Enable dumps from all OMP (Offloading and Multi Processing) optimizations.
vec	Enable dumps from all vectorization optimizations.
optall	Enable dumps from all optimizations. This is a superset of the optimization groups listed above.

  If *options* is omittet, it defaults to "optimized-optall". Some examples: "gcc -O3 -fopt-info", "gcc -O3 -ftree-vectorize -fopt-info-missed-vec", "gcc -fopt-info-loop-optimized-missed".

- -fpeel-loops
  Splits some problematic first or last loop iterations from a loop and performs them outside of the loop body. It also turns on complete loop peeling (i.e. complete removal of loops with small constant number of iterations).

- -fprefetch-loop-arrays
  Generates instructions to prefetch memory to improve the performance of loops that access large arrays. This option has only an effect if the target machine supports prefetching. Depending on the loop structure, this option may generate better or worse code.

- -freorder-blocks
  Reorders basic blocks in the compiled functions to reduce the number of branches and to improve code locality. This is the default if you compile with "-O2" or higher.

- -freorder-functions
  Reorders functions in the object file in order to improve code locality if that is possible. This is the default if you compile with "-O2" or higher.

- -ftree-loop-distribution
  Separates independent statements in the body of a single loop into separate loops with identical headers. This option can improve cache performance on big loop bodies and allows other loop optimizations like parallelization or vectorization.

- -ftree-loop-distribute-patterns
  Perform loop distribution of patterns that can be coded with calls to library functions. This is the default if you compile with "-O3".

```
for (i = 0; i < N; ++i)
{
  a[i] = 0;
  b[i] = a[i] + i;
}
```

Can be transformed to

```
for (i = 0; i < N; ++i)
{
  a[i] = 0;
}
for (i = 0; i < N; ++i)
{
  b[i] = a[i] + i;
}
```

Now the initialization loop can be transformed into a call to *memset* zero.

- -ftree-loop-linear
  Performs linear loop transformations. This option can improve cache performance and allows further loop optimizations to take place.

- -ftree-loop-optimize
  Performs loop optimizations. This is the default if you compile with "-O1" or higher.

- -ftree-loop-vectorize
  Performs loop vectorization on trees. This is the default if you compile with "-O3" and when "-ftree-vectorize" is enabled.

- -ftree-parallelize-loops=n
  Splits the iteration space of a loop so that *n* threads can do the computations in parallel. This is only possible if the instructions in the loop body are independent so that they can be reordered. The optimization is profitable for compute-intensive loops.

- -ftree-slp-vectorize
  Performs basic block vectorization. This is the default if you compile with "-O3" and when "-ftree-vectorize" is enabled.

- -ftree-vectorize
  Performs vectorization on trees. This flag enables "-ftree-loop-vectorize" and "-ftree-slp-vectorize" if not explicitly specified.

- -funroll-loops
  Unrolls loops if the number of iterations can be determined at compile time or upon entry to the loop. This option makes code larger and may or may not make it run faster.

- -funroll-all-loops
  Unrolls all loops , even if their number of iterations is uncertain when the loop is entered. This usually makes programs run more slowly.

- -funswitch-loops
  Moves branches with loop invariant conditions out of the loop with duplicates of the loop on both branches (modified according to result of the condition).

- -fvariable-expansion-in-unroller
  Creates multiple copies of some local variables when unrolling a loop, which can result in superior code.

- -march=[native | sandybridge | ivybridge | haswell | opteron | …]
  Produces code optimized for the most common IA32/AMD64/EM64T processors or the named special processor. Gcc 6.x will also support "skylake" and "skylake-avx512".

- -mfpmath=[387 | sse | …]
  Generates floating-point arithmetic for the standard floating-point coprocessor or uses the instructions of the SSE instruction set. For the i386 compiler you must also specify "-march=cpu-type", "-msse", or "-msse2" to enable SSE instructions and make this option effective. For the x86_64 compiler these extensions are enabled by default.

- -m[sse4 | avx | avx2 | avx512f | avx512pf | avx512er | avx512cd | …]
  Enables the "Streaming SIMD Extensions" or "Advanced Vector Extensions" instruction set on Intel/AMD processor architectures. Gcc 6.x will also support "avx512vl", "avx512bw", "avx512dq", "avx512ifma", and "avx512vbmi".

- optimization flags
  - improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program
  - if multiple "-O" options exist, the last option is the one that is effective
  - -O1
    Reduces the code size and execution time without performing any optimizations that take a great deal of compile time.
  - -O2
    Performs nearly all supported optimizations that do not need a compromise between space and speed.
  - -O3
    Turns on additional optimizations, e.g., "-finline-functions" and "-funswitch-loops".

- --param *name=value*
  - controls the amount of optimization that is done
  - *value* is always an *integer*
  - allowed choices for *name*
    - max-unroll-times
      The maximum number of unrollings of a single loop.
    - max-peel-times
      The maximum number of peelings of a single loop.
    - max-unswitch-level
      The maximum number of branches unswitched in a single loop.
    - max-block-tile-size
      Loop blocking or strip mining transformations strip mine each loop in the loop nest by a given number of iterations. The strip/block length can be changed with this parameter. The default value is 51.
    - …

- example

```
gcc mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":         51.00 s       51.33 s

gcc -O1 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          8.00 s        7.75 s

gcc -O2 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          8.00 s        7.85 s

gcc -O3 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        5.75 s

gcc -O3 -ftree-vectorize -floop-parallelize-all mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        5.88 s

gcc -O3 -ftree-vectorize -ftree-parallelize-loops=1 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        5.64 s

gcc -O3 -ftree-vectorize -ftree-parallelize-loops=2 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          4.00 s        7.95 s

gcc -O3 -ftree-vectorize -ftree-parallelize-loops=4 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          2.00 s        8.81 s

gcc -O3 -ftree-vectorize -ftree-parallelize-loops=8 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          1.00 s        9.08 s
```

A.2 Intel Parallel Studio XE 2017

- -check=keyword[, keyword, …]
  Checks some conditions at run-time. Optimizations are automatically disabled, if this option will be used (debug mode). Possible values for *keyword*:

[no]conversion	*conversion* checks if a conversion to a smaller type occurs
[no]stack	*stack* checks the stack frame for buffer overruns and buffer underruns. It also enforces stack pointer verification and local variables initialization.
[no]uninit	*uninit* checks for uninitialized local scalar variables. A run-time error occurs, if a variable is read before it is written. Read the manual page for more details.

  The default is that nothing will be checked at runtime.

- -check-pointers=keyword
  -check-pointers-mpx=keyword

  Checks bounds for memory access through pointers. *check-pointers-mpx* checks bounds for memory access through pointers on processors that support Intel Memory Protection Extensions. Possible values for *keyword*:

none	disables bound checking (default)
rw	checks bounds for reads and writes through pointers
write	checks bounds for only writes through pointers

- -falign-functions=[2 | 16]
  Aligns the start of functions on a 2 (default) or 16 byte boundary.

- -finline-functions
  Integrates all simple functions into their callers. The compiler heuristically decides which functions are simple enough for this operation. This is the default if you compile with “-O2”.

- -finline-limit=<n>
  Sets the number of statements a function can have and still be considerd for inlining.

- -[no-]fma
  Enables / disables the combining of floating-point multiplies and add / subtract operations (fused multiply-add instructions).

- -fopenmp
  Same as “-qopenmp”.

- - fp-model [fast={1 | 2} | precise | …]
  Enables floating-point model variation. “fast” (default: fast=1) enables more aggressive floating-point optimizations at a slight cost in accuracy or consistency and “precise” allows only value-safe optimizations.

- -funroll-loops
  Unrolls loops based on default heuristics.

- -guide[=level]
  Lets you set a level (1 - 4) of guidance for auto-vectorization, auto-parallelization, and data transformation. The default level is 4, when “level” is omitted. “4” is the most advanced and aggressive level. Auto-parallelization advice is given only if “-parallel” is also specified. The compiler doesn’t produce any objects or executables, when this option is specified.

- -guide-data-trans[=level]
  Lets you set a level (1 - 4) of guidance for data transformation. The default level is 4, when “level” is omitted. “4” is the most advanced and aggressive level.

- -guide-par[=level]
  Lets you set a level (1 - 4) of guidance for auto-parallelization. The default level is 4, when “level” is omitted. “4” is the most advanced and aggressive level. Auto-parallelization advice is given only if “-parallel” is also specified.

- -guide-vec[=level]
  Lets you set a level (1 - 4) of guidance for auto-vectorization. The default level is 4, when “level” is omitted. “4” is the most advanced and aggressive level.

- -inline-level=[0 | 1 | 2]
  Controls inline expansion. “0” disables inlining, “1” inlines functions declared with an inline keyword, and “2” inlines any function at the compilers’s discretion.

- -[no-]ip
  Enables (default) / disables single-file interprocedural optimization.

- -ipo[n]
  Enables multi-file interprocedural optimization. *n* specifies the number of object files the compiler should create. You cannot specify the names for the files that are created. Default for *n* is 0 (the compiler decides whether to create one or more object files). If *n* is greater than zero and less than the number of source files, the compiler creates *n* object files. Otherwise the compiler creates one object file for every source file.

- -march=[core2 | core-avx2 | corei7-avx | …]
  Generates code exclusively for a specific processor.

- -mgpu-arch=<arch>[,<arch>]
  Builds offload code for graphics to run on a particular graphics processor that is on the Intel microarchitecture code name <arch>. Possible values for <arch>: ivybridge, haswell.

- -mgpu-asm-dump[=<filename>]
  Generates a native assembly listing for the processor graphics code to be offloaded.

- -mkl{=[parallel | sequential | cluster]}
  Links to the Intel Math Kernel Library and bring in the associated headers. "-mkl" is equivalent to "-mkl=parallel" which uses the threaded MKL library. "sequential" uses the non-threaded MKL library and "cluster" uses the MKL cluster library plus the sequential MKL library.

- -mtune=[core2 | core-avx2 | corei7-avx | …]
  Optimizes code for a specific processor.

- -On specifies the optimization level

  – -O0
    Disables optimizations.

  – -O1
    Optimizes for size. Omits some optimizations which increase code size for a small speed benefit.Creates the smallest optimized code in most cases.

  – -O2 or -O
    Optimizes for maximum speed. This is the default setting. Enables many optimizations, including vectorization.

  – -O3
    Optimizes for maximum speed and enable more aggressive optimizations (e.g., scalar replacement, loop unrolling, code replication to eliminate branches, loop blocking, data prefetching) that may not improve performance on some programs.

  – -Ofast
    Enables "-O3 -no-prec-div -fp-model fast=2".

  – -fast
    "-Ofast" and additionally enables "-xHost -ipo -static".

- -parallel
  Enables the auto-parallelizer to generate multithreaded code for loops that can be safely executed in parallel.

- -par-threshold[n]
  Sets the threshold for the auto-parallelization of loops where *n* is an integer from 0 to 100 (default). “0” implies that loops will be parallelized regardless of computational work and “100” implies that loops will only be parallelized if a performance benefit is highly likely. Must be used in conjunction with “-parallel”.

- -[no-]prec-div
  Reduce / improve precision of floating-point divides (may have some speed impact).

- -qopenmp
  Enables explicit parallelization with OpenMP directives and defines the preprocessor token _OPENMP. You can set the environment variable OMP_NUM_THREADS prior to the execution of the parallelized program to specify how many threads should be used. You must set the environment variable OMP_NESTED to TRUE to enable nested parallelism.

- -q[no-]openmp-simd
  Enables / disables OpenMP SIMD compilation. Enabled by default with “-qopenmp”.

- -qopenmp-stubs
  Enables the user to compile OpenMP programs in sequential mode. The OpenMP directives are ignored and a sequential stub OpenMP library is linked.

- -qopt-block-factor=<n>
  Specifies preferred blocking factor for loop blocking (overriding default heuristics). Loop blocking is enabled at “-O3”.

- -q[no-]opt-dynamic-align
  Enables (default) / disables dynamic data alignment optimizations.

- -q[no-]opt-matmul
  Replaces matrix multiplication with calls to intrinsics and threading libraries for improved performance (default at “-O3 –parallel”). This option has no effect unless “-O2” or higher is set.

- -qopt-report[=n]
  Generates an optimization report at different levels. You can specify values 0 through 5. If you specify zero, no report is generated. The default level is 2. Use “-qopt-report-help” to learn which levels are available for different phases, if you want to select a special level.

- -qopt-report-phase=<phase>[,<phase>,…]
  Specify one or more phases that reports are generated against. Some commonly used values are as follows (“-qopt-report-help” displays all possible values).
  - all — all possible optimization reports for all phases (default)
  - loop — loop nest optimizations (unrolling, fusion, collapsing, ...)
  - openmp — OpenMP optimizations (reports among others the loops, regions, sections and tasks that were successfully parallelized).

par	auto-parallelizer optimizations
vec	vectorizer optimizations

- -[no-]scalar-rep
  Enables (default) / disables scalar replacement (requires -O2 or higher).

- -[no-]simd
  Enables (default) / disables vectorization using simd pragma.

- -static
  Prevents linking with shared libraries.

- -unroll[n]
  Sets maximum number of times to unroll loops. Omit *n* to use default heuristics. Use *n=0* to disable the loop unroller.

- -[no-]unroll-aggressive
  Enables / disables more aggressive unrolling heuristics.

- -[no-]vec
  Enables (default) / disables vectorization.

- -[no-]vec-guard-write
  Enables / disables cache / bandwidth optimizations for stores under conditionals within vector loops.

- -vec-threshold[n]
  Sets a threshold for the vectorization of loops based on the probability of performance gain. “0” vectorizes loops regardless of amount of computational work and “100” vectorizes loops only if a performance benefit is almost certain.

- -xHost
  Generates instructions for the highest instruction set and processor available on the compilation host machine.

- -x[SSE4.2 | AVX | CORE-AVX2 | CORE-AVX512 | COMMON-AVX512 | …]
  Enables the “Streaming SIMD Extensions” or “Advanced Vector Extensions” instruction set on Intel/AMD processor architectures.

- -Zp[1 | 2 | 4 | 8 | 16]
  Specifies alignment constraints for structures (default: -Zp16).

- example

```
icc -O0 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":         42.00 s       41.78 s

icc -O1 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          9.00 s        8.95 s

icc -O2 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        6.22 s

icc -O3 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          4.00 s        3.37 s

icc -Ofast mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          4.00 s        4.15 s

icc -Ofast -xHost -ipo mat_mult_ikj.c
icc: command line remark #10382: option '-xHOST' setting '-xCORE-AVX2'
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          0.00 s        0.86 s

icc -Ofast -xCORE-AVX2 -ipo -DP=4096 -DQ=4096 -DR=4096 mat_mult_ikj.c
a.out
c[4096][4096] = a[4096][4096] * b[4096][4096] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          9.00 s        9.06 s

icc -Ofast -xCORE-AVX2 -ipo -qopt-matmul -DP=4096 -DQ=4096 -DR=4096 mat_mult_ikj.c
a.out
c[4096][4096] = a[4096][4096] * b[4096][4096] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          1.00 s        9.11 s

icc -Ofast -xCORE-AVX2 -ipo -parallel -DP=4096 -DQ=4096 -DR=4096 mat_mult_ikj.c
setenv OMP_NUM_THREADS 4
a.out
c[4096][4096] = a[4096][4096] * b[4096][4096] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          1.00 s        4.68 s
```

A.3 LLVM clang 5.x

- --cuda-compile-host-device
  Compile CUDA code for both host and device. This is the default.
- --cuda-device-only
  Compile CUDA code for device only.
- --cuda-host-only
  Compile CUDA code for host only.
- --cuda-noopt-device-debug
  Enable device-side debug information generation, i.e. disables *ptxas* optimizations.
- --cuda-path=<path>
  Specify the path to the CUDA installation, if the CUDA SDK isn't available in */usr/local/cuda*, */usr/local/cuda-7.0*, */usr/local/cuda-7.5*, or */usr/local/cuda-8.0*. Maybe you must also pass *-L<CUDA library path>* and *-l<CUDA lib>* to link a program.
- --[no-]cuda-gpu-arch=<arch>
  Specify a CUDA GPU architecture, i.e. the compute capability, of a GPU for which code should be generated. Possible values for *arch* are sm_30, sm_50, … This option can appear more than once on the command line, if code for different architectures must be created. You can remove a GPU architectures from the list of GPUs to compile for with "--no-cuda-gpu-arch" and you can reset the list to its default with "--no-cuda-gpu-arch=all".
- --no-cuda-version-check
  Don't error out if the detected version of CUDA is too low for the requested CUDA GPU architecture.
- --ptxas-path=<path>
  Path to the **p**arallel **t**hread e**x**ecution **as**sembler (ptxas) which is needed for compiling CUDA code, if the CUDA SDK isn't available in one of the above mentioned directories.
- -cl-std=<std>
  OpenCL language standard to compile for. Possible values for *std* are CL1.1, CL1.2, and CL2.0. Many other build options from the "OpenCL specification v2.0 section 5.8.4" are also available.
- -finline-functions
  Integrates all simple functions into their callers. The compiler heuristically decides which functions are simple enough for this operation. This is the default if you compile with "-O3".
- -malign-double
  Align *double* to two words in structures.

- -mllvm <option>
  Additional arguments to forward to LLVM's option processing. This option can appear multiple times on the command line. Possible values for *option*:
  - -force-vector-width=<value>
    Controls the vectorization SIMD width.
  - -force-vector-interleave=<value>
    Controls the unroll factor.
  - -polly
    Optimize with Polly.
  - -polly-parallel
    Automatically detects parallel loops and generates OpenMP code for them. You must also add “-mllvm -polly" to activate Polly and “-lomp” to link with the OpenMP library.
  - -polly-show, -polly-show-only, -polly-dot, -polly-dot-only
    Polly can show the SCoPs (**S**tatic **Co**ntrol **P**arts) of a program with *graphviz*. “show” options automatically run a *graphviz* viewer and “dot” options store for each function a dot file that highlights the detected SCoPs. If “only” is appended at the end of the option, the basic blocks are shown without the statements the contain. Example:

    ```
    clang –O3 –mllvm –polly –mllvm –polly-vectorizer=stripmine –mllvm
    –polly-dot-only mat_mult_ikj.c
    ```

    ```
    dot –Tgif scopsonly.main.dot –o scops.gif
    ```
  - -polly-vectorizer=<strategy>
    Selects a vectorization strategy for Polly. You must also add “-mllvm -polly" to activate Polly. Possible values for *strategy*:
    - ♦ none — no vectorization
    - ♦ polly — Polly internal vectorizer
    - ♦ stripmine — strip-mine outer loops for the loop-vectorizer

- -f[no-]openmp
  Enables / disables (default) explicit parallelization with OpenMP directives and defines the preprocessor token _OPENMP if *-fopenmp* is specified. You can set the environment variable OMP_NUM_THREADS prior to the execution of a parallelized program to specify how many threads should be used.

- -f[no-]reroll-loops
  Enables / disables loop rerolling (the opposite of loop unrolling). Loop rerolling makes code smaller and may or may not make it run slower.

- -f[no-]slp-vectorize
  Enables (default) / disables the SLP (superword-level parallelism) vectorizer which combines similar independent instructions into vector instructions, e.g. memory accesses, arithmetic operations, or comparison operations can be vectorized using this technique.

- -f[no-]slp-vectorize-aggressive
  Enables / disables the BB (basic block) vectorizer which is more compile-time intensive.

- -f[no-]unroll-loops
  Enables / disables loop unrolling. This option makes code larger and may or may not make it run faster.

- -f[no-]vectorize
  Enables (default) / disables the loop vectorizer which uses a cost model to decide on the optimal vectorization and unroll factors. However, users can force the vectorizer to use specific values with "-mllvm -force-vector-width" and "-mllvm -force-vector-interleave".

- -fopenmp-targets=<target>[,<target>[,…]]
  Specify a comma-separated list of target triples for supported OpenMP offloading targets. A possible value for *target* is "nvptx64-nvidia-cuda". A target triple has the general format "<arch><sub>-<vendor>-<sys>-<abi>" where the different components can have the following values.
  - arch x86_64, nvptx64, arm, etc.
  - sub for example on ARM: v5, v6m, etc.
  - vendor unknown, apple, ibm, nvidia, etc.
  - sys none, cuda, darwin, linux, win32, etc.
  - abi android, elf, gnu, etc.

  Sub-architecture options depend on the architectures, so that "x86v5" wouldn't make sense. Most of the time it can be omitted, which sets the defaults for the specified architecture. The system name *sys* is generally the operating system (darwin, linux), but could be special like the bare-metal "none". When a parameter is not important, it can be omitted, or you can choose *unknown* and the defaults will be used. If you choose a parameter that *clang* doesn't know, it will be ignored and assumed as *unknown*. The target triple for our Linux machines is "x86_64-unknown-linux-gnu".

- -On specifies the optimization level
  - -O0
    Disables optimizations.
  - -O1
    Optimize for fast execution without triggering significant incremental compile time or code size growth.
  - -O2
    Optimize as much as possible for fast execution. This mode is significantly more aggressive in trading off compile time and code size to get execution time improvements.
  - -O3
    Optimize as much as possible for small code size.

- -Os
  Optimize for code size at any and all costs. You should expect this level to produce rather slow, but very small, code.
- -Oz
  Like "-Os" but with even more optimizations for size.

- -Rpass=<value>
  Report transformations performed by optimization passes whose names match the given POSIX regular expression. Possible values for *value*:
  - \.* or ".*" report all optimizations from all passes (the escape characters "\" or the quotation marks are necessary to prohibit that the "*command line shell*" will interpret "." and "*")
  - inline optimizations from the function inlining pass
  - gvn optimizations from the GVN pass (global value numbering)
  - licm optimizations from the LICM pass (loop invariant code motion)
  - polly-detect optimizations from the Polly Detect SCoPs pass (detects **S**tatic **Co**ntrol **P**arts in functions)
  - polly-scops optimizations from the Polly SCoPs pass (creates polyhedral description of SCoPs)
  - tailcallelim optimizations from the tail call elimination pass
  - …

  (Visit http://llvm.org/docs/Passes.htm, http://polly.llvm.org/documentation/passes.html, or https://en.wikipedia.org/wiki/Polytope_model for more information.)

- -Rpass-analysis=<value>
  Report when an optimization pass, whose name match the given POSIX regular expression, determines whether or not to make a transformation. All values mentioned by "-Rpass" are allowed.

- -Rpass-missed=<value>
  Report missed transformations by optimization passes whose names match the given POSIX regular expression. All values mentioned by "-Rpass" are allowed.

- -Xclang <arg>
  Pass *arg* to the *clang* compiler. This option can appear multiple times on the command line.

- -Xcuda-ptxas <arg>
  Pass *arg* to the *ptxas* assembler. This option can appear multiple times on the command line.

- -Xlinker<arg>
  Pass *arg* to the linker. This option can appear multiple times on the command line.

- example

loki introduction 116 **clang -O0 mat_mult_ikj.c**

loki introduction 117 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":         27.00 s       27.67 s
```

loki introduction 118 **clang -O1 mat_mult_ikj.c**

loki introduction 119 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          7.00 s        7.89 s
```

loki introduction 120 **clang -O2 mat_mult_ikj.c**

loki introduction 121 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        5.45 s
```

loki introduction 122 **clang -O3 mat_mult_ikj.c**

loki introduction 123 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        5.52 s
```

loki introduction 124 **clang -Os mat_mult_ikj.c**

loki introduction 125 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        5.77 s
```

loki introduction 126 **clang -Oz mat_mult_ikj.c**

loki introduction 127 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          8.00 s        8.16 s
```

loki introduction 128 **clang -O2 -mllvm -polly mat_mult_ikj.c**

loki introduction 129 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          2.00 s        2.16 s
```

loki introduction 130 **clang -O2 -mllvm -polly -mllvm -polly-vectorizer=polly mat_mult_ikj.c**

loki introduction 131 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          2.00 s        2.03 s
```

loki introduction 132 **clang -O2 -mllvm -polly -mllvm -polly-vectorizer=stripmine mat_mult_ikj.c**

loki introduction 133 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          2.00 s        1.99 s
```

loki introduction 134 **clang -O2 -mllvm -polly -mllvm -polly-parallel mat_mult_ikj.c**

loki introduction 135 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          2.00 s        2.16 s
```

loki introduction 136 **clang -O2 -mllvm -polly -mllvm -polly-parallel -mllvm -polly-vectorizer=polly mat_mult_ikj.c**

loki introduction 137 **a.out**

```
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          2.00 s        2.05 s
```

loki introduction 138 **clang -O2 -mllvm -polly -mllvm -polly-parallel -mllvm -polly-vectorizer=stripmine mat_mult_ikj.c -lomp**

loki introduction 139 **a.out**

```
loki introduction 140 a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          2.00 s        2.01 s
```

A.4 Microsoft Visual Studio 2017

- /arch:[AVX | AVX2]
  Use AVX or AVX2 instructions for code generation.

- /GF
  Eliminate duplicate strings, i.e., the compiler creates only a single copy of identical strings. This optimization is called string pooling and can create smaller programs. String pools are read-only, i.e., an error occurs, if the program tries to modify a string in the pool.

- /GL[-]
  Enable whole program optimization. Optimizes the use of registers across function boundaries, allows a reduction in the number of load and store operations for global data, inlines functions even when the function is defined in another module, and so on. Optimizations are only performed for each module without "/GL". Whole program optimization is not enabled by default and it can be explicitly disabled with "/GL-".

- /GS[-]
  Detects some buffer overruns. Buffer overrun is a technique used by hackers to exploit code that does not check for buffer overruns. "/GS" is enabled by default and can be disabled with "/GS-".

- /Gs[size]
  Controls stack checking calls for every function call that requires storage for local variables. Local variables can occupy "size" bytes (default: one page, 4 KB) before a stack probe is activated. The default value of "size" allows the program stack to properly grow at run time.

- /Gw[-]
  Optimize whole program global data. Global data optimization is not enabled by default and can be explicitly disabled with "/GL-".

- /Gy
  Enable function-level linking.

- /O1
  Minimize program size. "/O1" is a shortcut for "/GF /Gs /Gy /Ob2 /Og /Oy /Os".

- /O2
  Maximize speed. "/O2" is a shortcut for "/GF /Gs /Gy /Ob2 /Og /Oy /Oi /Ot".

- /Ob[0 | 1 | 2]
  Enables the compiler to inline functions. "0" disables inline function expansion, "1" expands only functions marked with "inline" and similar keywords, and "2" enables "auto-inlining" so that the compiler can choose which functions should be expanded. This option requires that "/O1", "/O2", "/Og", or "/Ox" is enabled.

- /Od
  Disables all optimizations. "/Od" is the default, so that debugging is easier.

- /Og
  Provides local, global, and loop optimizations. This option is deprecated and should be replaced by "/O1" (optimize for size) or "/O2" (optimize for speed).

- /Oi[-]
  Enables the compiler to replace some function calls with intrinsic functions, which are faster because they do not have the overhead of function calls. "/Oi-" disables intrinsic functions.

- /Os
  Favors small code. You must also specify "/Og" to optimize the code.

- /Ot
  Favors fast code. You must also specify "/Og" to optimize the code.

- /Ox
  Enables the compiler to use maximum optimization (favoring speed over code size). This option is a shortcut for "/Ob2 /Og /Oi /Ot /Oy".

- /Oy
  Suppresses the creation of frame pointers on the call stack which accelerates function calls, because it is not necessary to set up and remove frame pointers.

- /openmp
  Enables explicit parallelization with OpenMP directives and defines the preprocessor token _OPENMP. You can set the environment variable OMP_NUM_THREADS prior to the execution of the parallelized program to specify how many threads should be used. You must set the environment variable OMP_NESTED to TRUE to enable nested parallelism.

- /Qpar[-]
  Enables ("/Qpar-" disables) the auto-parallelizer, so that the compiler can automatically parallelize loops. A parallelization takes place if it is legal and would improve performance. The following directives are available to help the optimizer to parallelize specific loops.

  – #pragma loop(hint_parallel(n))

    Parallelize loop across *n* threads ($n \geq 0$). Use the maximum number of threads at run time, if "n == 0".

  – #pragma loop(no_vector)

    Disable the auto-vectorizer for the following loop, which is enabled by default and tries to vectorize all loops.

  – #pragma loop(ivdep)

    Ignore vector dependencies for the following loop. Use this directive in conjunction with "hint_parallel".

- /Qpar-report:{1 | 2}
  Enables the reporting feature for the auto-parallelizer. “1” outputs a message for loops that are parallelized and “2” outputs a message for loops that are parallelized and also for loops that are not parallelized (together with a reason code).

- /Qvec-report:{1 | 2}
  Enables the reporting feature for the auto-vectorizer. 1” outputs a message for loops that are vectorized and “2” outputs a message for loops that are vectorized and also for loops that are not vectorized (together with a reason code).

- /Zp[1 | 2 | 4 | 8 | 16]
  Specifies alignment constraints for structures (default: /Zp8).

- example

```
cl mat_mult_ikj.c
mat_mult_ikj.exe
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":         28.00 s       28.23 s


cl /GL /Gw /Ox /Qpar /Qpar-report:2 /Qvec-report:2 mat_mult_ikj.c
...
--- Analyzing function: main
mat_mult_ikj.c(107) : info C5002: Loop not vectorized due to '1300'
mat_mult_ikj.c(105) : info C5002: Loop not vectorized due to '1106'
...
mat_mult_ikj.c(137) : info C5001: Loop vectorized
...
mat_mult_ikj.c(151) : info C5002: Loop not vectorized due to '1304'
...
mat_mult_ikj.c(107) : info C5012: Loop not parallelized due to '1008'
...
mat_mult_ikj.c(135) : info C5012: Loop not parallelized due to '1000'
...

mat_mult_ikj.exe
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          7.00 s        6.84 s
```

reason	description
1000	data dependency in the loop body
1008	not enough work to warrant auto-parallelization
1106	outer loop not vectorized
1300	loop body contains no – or very little - computation
1304	loop includes assignments that are of different sizes

A complete list of vectorizer and parallelizer messages is available at
https://msdn.microsoft.com/en-us/library/jj658585.aspx (english) and
https://msdn.microsoft.com/de-de/library/jj658585.aspx (deutsch).

A.5 Oracle Developer Studio 12.x

- -fast
  Tunes an executable for maximum run-time performance. Use the options "-#" or "-xdryrun" to examine the expansion of the macro "-fast".

- -fma=[none | fused]
  Enables automatic generation of floating-point fused multiply-add instructions. The default is "-fma=none".

- -fopenmp
  Equivalent to "-xopenmp=parallel".

- -fsimple=[0 | 1 | 2]
  "fsimple=2" enables use of SIMD instructions to compute reductions when "-xvector=simd" is in effect. This is the default if you compile with "-fast".

- -library=sunperf
  Links with the *Oracle Developer Studio* supplied performance library.

- -mt [={yes | no}]|
  Use this option to compile and link multithreaded code. "-mt=yes" assures that libraries are linked in the appropriate order and that "-D_REENTRANT" is passed to the preprocessor. "-xopenmp" automatically includes "-mt=yes". "-mt" is equivalent to "-mt=yes" and is the default for the compiler.

- -xarch=[sse4_2 | avx | avx2 | …]
  Enables the "Streaming SIMD Extensions" or "Advanced Vector Extensions" instruction set on Intel/AMD processor architectures.

- -xautopar
  Turns on automatic parallelization for multiple processors / cores. Needs optimization level "-xO3". Does not accept OpenMP parallelization directives, so that you should avoid this option if you do your own thread management. You must set the environment variable OMP_NUM_THREADS prior to the execution of the parallelized program.

- -xbuiltin=[%all | %none]
  "%all" means that the compiler should substitute intrinsic or inline standard library functions to improve the performance. Intrinsic functions encapsulate processor specific instructions within a function. "-xbuiltin" only inlines global functions defined in system header files and never static functions defined by the user (you can use the option "-xO4" or "-xO5" to inline user functions). "%all" is the default if you compile with "-fast".

- -xcache={generic | native | s1/l1/a1[/t1] | s1/l1/a1[/t1]:s2/l2/a2[/t2] | s1/l1/a1[/t1]:s2/l2/a2[/t2] :s3/l3/a3[/t3]}
  "si" defines the size of the data cache at level "i" in kilobytes, "li" defines the cache line size of the data cache at level "i" in bytes, "ai" defines the associativity of the data cache at level "i" where "ai > 1" means n-way associativity and "ai = 1" means "direct mapped", and "ti" specifies the number of hardware threads sharing the cache at level "i". "ti" is optional and a value of "1" is used if not present.

- -xchip=[sandybridge | ivybridge | haswell | westmere | …]
  Specifies the target processor for the optimization. Effects the ordering of instructions, the way the compiler uses branches, and more.

- -xdepend=[yes | no]
  Analyses loops for inter-iteration dependencies and performs loop restructuring. Needs optimization level "-xO3". "yes" is the default if you compile with "-fast". The optimization will be done for a multiprocessor system if "-xautopar" is also defined.

- -xF {=[no%]func | [no%]glbdata | %all | %none}
  Enables/disables the optimal reordering of functions and variables by the linker. "%func" fragments functions into separate sections. "%glbdata" fragments global data (variables with external linkage) into separate sections. "%all" fragments functions and global data into separate sections. "%none" fragments nothing.

- -xipo {=[0 | 1 | 2]}
  Performs whole-program optimizations by invoking the interprocedural analysis component. "-xipo=0" is the default setting and turns off "-xipo". The compiler performs inlining across all source files with "-xipo=1" and additionally interprocedural aliasing analysis and optimization of memory allocation and layout to improve cache performance with "-xipo=2". "-xipo" is equivalent to "-xipo=1".

- -xlibmil
  Enables the compiler to inlines some math library functions for faster execution. This is the default if you compile with "-fast".

- -xlibmopt
  Enables the compiler to use a library of optimized math functions for faster execution. You must also specify "-fround=nearest". This is the default if you compile with "-fast".

- -xloopinfo
  Shows which loops are parallelized and which are not. Normally this option will be used together with "-xautopar".

- -xOn specifies the optimization level
  (The particular kind of optimization depends on the architecture of the processor.)
  - -xO1
    Does only basic local optimization (peephole).
  - -xO2
    Does basic local and global optimization (common subexpression elimination, algebraic simplification, register allocation, …).
  - -xO3
    Does additionally to "-xO2" loop unrolling, reference optimization, etc.
  - -xO4
    Does automatic inlining of functions contained in the same file, all optimizations of "-xO3", etc. In general, this level results in increased code size.
  - -xO5
    The highest level of optimization and the default if you compile with "-fast".
- -xopenmp {=[parallel | noopt | none]}
  Enables explicit parallelization with OpenMP directives and defines the preprocessor token _OPENMP. "-xopenmp" is equivalent to "-xopenmp=parallel", enables OpenMP pragmas, and changes the optimization level to "-xO3" if necessary. "-xopenmp=noopt" enables OpenMP pragmas but doesn't raise the optimization level if it is lower than "-xO3". It issues an error if you explicitly set the optimization level lower than "-xO3". "-xopenmp=none" is the default. You can set the environment variable OMP_NUM_THREADS prior to the execution of the parallelized program to specify how many threads should be used. You must set the environment variable OMP_NESTED to TRUE to enable nested parallelism. Do not use this option together with "-xautopar".
- -xpagesize=[default | 4K | 2M | 4M | 1G]
  Sets the preferred page size for the stack and the heap. The above values are for x86/x86_64 platforms.
- -xpagesize_heap=[default | 4K | 2M | 4M | 1G]
  Sets the preferred page size for the heap. The above values are for x86/x86_64 platforms.
- -xpagesize_stack=[default | 4K | 2M | 4M | 1G]
  Sets the preferred page size for the stack. The above values are for x86/x86_64 platforms.
- -xreduction
  Analyses loops for reduction in automatic parallelization. Can only be used in combination with "-xautopar".

- -xtarget=[generic64 | native64 | sandybridge | haswell | westmere | …]
  Permits a quick and easy specification of "-xarch", "-xchip", and "-xcache" combinations that occur on real systems. You can change any of the settings by following "-xtarget" with a different value for "-xarch" and so on.

- -xunroll=n
  Specifies whether the compiler unrolls loops. If "n = 1" it doesn't unroll loops and if "n > 1" it allows the compiler to unroll loops n times.

- -xvector {=[no%]simd | %none | …}
  "simd" enables the compiler to use the native x86 SSE SIMD instructions to improve performance of certain loops. This is the default on x86/x86_64 platforms. "-xvecor" is equivalent to "-xvector=simd".

- -xvpara
  Show parallelization warning messages about potential parallel programming related problems that may cause incorrect results when using OpenMP.

- example

```
cc -xO1 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":         17.00 s       16.70 s

cc -xO2 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          8.00 s        8.29 s

cc -xO3 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        5.58 s

cc -xO4 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        5.56 s

cc -xO5 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          5.00 s        5.57 s

cc -fast mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          8.00 s        8.08 s

cc -fast -xautopar -xreduction -xloopinfo mat_mult_ikj.c
"mat_mult_ikj.c", line 105: PARALLELIZED, fused
"mat_mult_ikj.c", line 125: PARALLELIZED
"mat_mult_ikj.c", line 133: PARALLELIZED
"mat_mult_ikj.c", line 135: not parallelized, not profitable, interchanged
"mat_mult_ikj.c", line 137: not parallelized, not profitable, interchanged
"mat_mult_ikj.c", line 149: PARALLELIZED, reduction

setenv OMP_NUM_THREADS 1
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          7.00 s        6.73 s

setenv OMP_NUM_THREADS 6
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          1.00 s        5.35 s
```

A.6 Portland Group compiler (Community Edition 2016)

- -acc[=options]
  Enable explicit parallelization with OpenACC directives and define the preprocessor token _OPENACC. See the *-ta* flag to select target accelerators for which to compile. The options are:

[no]autopar	Enable loop autoparallelization within parallel constructs. The default is *autopar*.
[no]routineseq	Compile every routine for the device, as if it had a routine *seq* directive. The default is *noroutineseq*.
strict	Issue warnings for non-OpenACC accelerator dirctives.
verystrict	Fail with an error for non-OpenACC accelerator dirctives.
sync	Ignore async clauses, and run every data transfer and kernel launch on the default sync queue.
[no]wait	Wait for each compute kernel to finish. The default is *nowait*.

- -fast
  Choose generally optimal flags for the target platform (includes -O2 -Munroll=c:1 -Mnoframe -Mlre -Mautoinline)

- -fastsse
  Same as “-fast”.

- -M[no]autoinline[=option[,option[,option]]]
  Enables automatic function inlining. Default: -Mnoautoinline. The options are:

level:<n>	automatically inline up to *n* levels of function calls (default: 10 levels)
maxsize:<n>	automatically inline functions up to a size of *n* (default: roughly 100 statements)
totalsize:<n>	limit automatical inlining to a total size of *n* (default: roughly 8.000 statements)

- -Mcache_align
  Align unconstrained data objects of size greater than or equal to 16 bytes on cache-line boundaries. An unconstrained object is a variable or an array that is not a member of an aggregate structure, is not allocatable, and is not an automatic array.

- -Mconcur[=option[,option,…]]

  Enable automatic parallelization for multiple processors / cores. Needs at least optimization level "-O2". The environment variable OMP_NUM_THREADS controls how many threads will be used to execute parallelized loops. The options can be one or more of the following:

allcores	Use all available cores when the environment variable OMP_NUM_THREADS isn't set.
bind	Bind threads to cores or processors.
[no]altcode:n	Generate (don't generate) alternate scalar code for parallelized loops. The parallelizer generates scalar code to be executed whenever the loop count is less than or equal to *n*. If *noaltcode* is specified, the parallelized version of the loop is always executed regardless of the loop count.
altreduction[:n]	Generate alternate scalar code for parallelized loops containing a reduction. If a parallelized loop contains a reduction, the parallelizer generates scalar code to be executed whenever the loop count is less than or equal to *n*.
[no]assoc	Enable (disable) parallelization of loops with reductions. The default is *assoc*.
[no]cncall	Assume (don't assume) that loops containing calls are safe to parallelize. The default is *nocncall*.
dist:[block | cyclic]	Parallelize with block or cyclic distribution. Contiguous blocks of iterations of a parallelizable loop are assigned to the available processors with *block* distribution. With *cyclic* distribution the outermost parallelizable loop in any loop nest is parallelized. If a parallelized loop is innermost, its iterations are allocated to processors cyclically. For example, if there are 3 processors executing a loop, processor 0 performs iterations 0, 3, 6, etc; processor 1 performs iterations 1, 4, 7, etc; and processor 2 performs iterations 2, 5, 8, etc.
[no]innermost	Enable (disable) parallelization of innermost loops. The default is *noinnermost*.
levels:n	Parallelize loops nested at most *n* levels deep. The default is 3.
[no]numa	Use (don't use) thread / processor affinity for NUMA architectures. "-Mconcur=numa" will link in a numa library and objects to prevent the operating system from migrating threads from one processor to another.

- -M[no]fma

  Generate (don't generate) fused multiply-add (FMA) instructions for targets that support it. FMA instructions are generally faster than separate multiply-add instructions, and can generate higher precision results since the multiply result is not rounded before the addition. However, because of this, the result may be different than the unfused multiply and add instructions. FMA instructions are enabled with higher optimization levels. The default is *-Mnofma*.

- -M[no]frame
  Set up (don't set up) a true stack frame pointer for functions. *-Mnoframe* allows slightly more efficient operation when a stack frame is not needed, but some options override *-Mnoframe*. The default is *-Mnoframe*.

- -Minfo[=option[,option,…]]
  Emit useful information to *stderr*. The options can be one or more of the following:

all	Includes options *accel*, *inline*, *ipa*, *loop*, *lre*, *mp*, *opt*, *par*, *unified*, and *vect*. This is the default for *-Minfo* without options.
accel	Emit information about accelerator region targeting.
inline	Emit information about functions extracted and inlined.
intensity	Emit compute intensity information about loops. Compute intensity is defined as the ratio of the number of arithmetic operations performed on computed data divided by the number of memory transfers required to move that data between two levels of a memory hierarchy.
ipa	Emit information about the optimizations enabled by interprocedural analysis (IPA).
loop | opt	Emit information about loop optimizations. This includes information about vectorization and loop unrolling.
lre	Emit information about loop-carried redundancy elimination.
mp	Emit information about OpenMP parallel regions.
par	Emit information about loop parallelization.
pfo	Emit profile feedback information.
time | stat	Emit compilation statistics.
unified	Emit information about which routines are selected for target-specific optimizations using the PGI Unified Binary.
vect	Emit information about automatic loop vectorization.

- -M[no]lre[=assoc | noassoc]
  Enable (disable) loop-carried redundancy elimination. The *assoc* option allows expression reassociation, and the *noassoc* option disallows expression reassociation.

- -M[no]unroll[=option[,option,…]]
  Enable (disable) loop unrolling. Needs at least optimization level “-O2”. The default is *-Mnounroll*. The options can be one or more of the following:

c:<u>	Completely unroll loops with a loop count less than or equal to *u*.
n:<u>	Unroll single-block loops *u* times.
m:<u>	Unroll multi-block loops *u* times.

- -M[no]vect[=option[,option,…]]
  Enable (disable) automatic vectorization. The default is *-Mnovect*. Needs at least optimization level "-O2". If no option list is specified, then the following vector optimizations are used: *assoc,cachesize:c,nosimd*, where *c* is the actual cache size of the machine. The options can be one or more of the following:

[no]altcode	Enable (disable) alternate code generation for vector loops, depending on such characteristics as array alignments and loop counts. The default is *altcode*.
[no]fuse	Enable (disable) loop fusion to combine adjacent loops into a single loop. The default is *nofuse*.
prefetch	Use prefetch instructions in loops where profitable.
[no]simd[:128|256]	Use vector SIMD instructions (SSE, AVX) instructions. The argument may be used to limit usage to 128-bit SIMD instructions. Specifying 256-bit SIMD instructions is only possible for target processors that support AVX. The default is *nosimd*.
[no]uniform	Perform the same optimizations in the vectorized and residual loops. This may affect the performance of the residual loop. The default is *nouniform*.

- -mp[=option]
  Enable explicit parallelization with OpenMP directives and define the preprocessor token _OPENMP. The options can be one or more of the following:

[no]align	Modify (don't modify) default loop iteration scheduling to align iterations with array references. The default is *noalign* and to use simple static scheduling.
allcores	Use all available cores when the environment variable OMP_NUM_THREADS isn't set.
bind	Bind threads to cores or processors.
[no]numa	Use (don't use) libraries to give affinity between threads and processors; this is useful with NUMA (non-uniform memory access) parallel architectures, so memory allocated by a particular thread will be allocated close to that processor, and will remain close to that thread. The default depends on the host machine.

- -O*n* specifies the optimization level
  If -O is not specified, then the default level is 1 if *-g* is not specified, and 0 if *-g* is specified. If a number is not supplied with -O then the optimization level is set to 2. The optimization levels and their meanings are as follows:

  – -O0
    A basic block is generated for each statement. No scheduling is done between statements. No global optimizations are performed.

- -O1
  Scheduling within extended basic blocks is performed. No global optimizations are performed.
- -O2
  All level 1 optimizations are performed. In addition, traditional scalar optimizations such as induction recognition and loop invariant motion are performed by the global optimizer. More advanced optimizations such as SIMD code generation, cache alignment and partial redundancy elimination are also enabled.
- -O3
  All -O1 and -O2 optimizations are performed. In addition, this level enables more aggressive code hoisting and scalar replacement optimizations that may or may not be profitable.
- -O4
  All -O1, -O2, and -O3 optimizations are performed. In addition, hoisting of guarded invariant floating point expressions is enabled.

- -ta=<target>
  Specify the type of the accelerator to which to target accelerator regions. *target* can be specified among others by the following values:

tesla[:options]	Compile the accelerator regions for a CUDA-enabled NVIDIA GPU. Options valid after "-ta=tesla" are among others:	
	cc50	Compile for compute capability 5.0.
	cc60	Compile for compute capability 6.0.
	cuda7.5	Use CUDA 7.5 Toolkit compatability.
	cuda8.0	Use CUDA 8.0 Toolkit compatability.
	loadcache:{L1 | L2}	Generate code to cache global memory loads in the L1 or L2 hardware cache.
	kepler	Generate code for a Kepler device.
	maxwell	Generate code for a Maxwell device.
	[no]unroll	Automatically (do not) unroll inner loops. This is enabled by default at optimization level *-O3*.
	The default is to generate code for compute capabilities 2.0 through 5.0, unless cuda8.0 is specified, in which case the default includes compute capability 6.0 as well. Specifying cc60 also implies the cuda8.0 option. Note that multiple compute capabilities can be specified, and one version will be generated for each capability specified. The default is equivalent to "-ta=tesla:fermi+".	
nvidia[:options]	This flag is equivalent to "-ta=tesla[:options]", and has all the same options.	
radeon[:options]	Compile the accelerator regions for an AMD Radeon GPU. Options valid after "-ta=radeon" are among others:	
	capverde	Generate code for Cape Verde architecture GPUs.
	spectre	Generate code for Spectre architecture GPUs.
	The default is equivalent to "-ta=radeon:tahiti".	

host	Compile the accelerator regions to run sequentially on the host processor.

The default in the absence of the -ta flag is to ignore the accelerator directives and compile for the host. Multiple targets are allowed, such as "-ta=tesla,host", in which case code is generated for the NVIDIA GPU as well as the host for each accelerator region.

- -tp=<target>

Specify the type of the target processor. *target* can be specified among others by the following values:

k8	AMD64 processor
barcelona	AMD Barcelona processor
bulldozer	AMD Bulldozer processor
p7	Intel 64 processor
core2	Intel core2 processor
sandybridge	Intel SandyBridge architecture Core processor
haswell	Intel Haswell architecture processor
x64	Unified AMD / Intel mode that is equivalent to -tp=k8,p7

The default in the absence of the *-tp* flag is to compile for the type of CPU on which the compiler is running.

- example

```
pgcc -O1 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":         19.00 s       19.02 s

pgcc -O2 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          6.00 s        6.20 s
```

("-O3" or "-O4" don't reduce the execution times again)

```
pgcc -O3 -Mconcur=allcores mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          0.00 s       16.58 s

setenv OMP_NUM_THREADS 12
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          0.00 s        9.71 s

pgcc -O3 -Mvect=simd:256 mat_mult_ikj.c
a.out
c[1984][1984] = a[1984][1984] * b[1984][1984] was successful.
                      elapsed time      cpu time
Mult "a" and "b":          5.00 s        5.60 s
```

A.7 Examples for compiler options

1. sequential program (GNU gcc)

```
gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-
prototypes -o mat_mult_ijk_gcc mat_mult_ijk.c
```

2. sequential program (Intel icc)

```
icc -std=c11 -m64 -fast -prec_div -Wall -Wcheck -o mat_mult_ijk_icc
mat_mult_ijk.c
```

3. sequential program (LLVM clang)

```
clang -std=c11 -m64 -O2 -o mat_mult_ijk_clang mat_mult_ijk.c
```

4. sequential program (Microsoft Visual Studio cl)

```
cl /GL /Gw /Ox /Femat_mult_ijk_cl.exe mat_mult_ijk.c
```

5. sequential program (Oracle Developer Studio cc)

```
cc -std=c11 -m64 -fast -fd -v -o mat_mult_ijk_cc mat_mult_ijk.c
```

6. sequential program (Portland Group pgcc)

```
pgcc -c11 -m64 -fast -Minfo -o mat_mult_ijk_pgcc mat_mult_ijk.c
```

7. automatic parallelization (GNU gcc)

```
gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-
prototypes -floop-parallelize-all -o mat_mult_ijk_auto_gcc mat_mult_ijk.c
```

8. automatic parallelization (Intel icc)

```
icc -std=c11 -m64 -fast -prec_div -Wall -Wcheck -parallel
-o mat_mult_ijk_auto_icc mat_mult_ijk.c
```

9. automatic parallelization (LLVM clang)

```
clang -std=c11 -m64 -O2 -mllvm -polly -mllvm -polly-parallel -mllvm
-polly-vectorizer=stripmine -o mat_mult_ijk_auto_clang mat_mult_ijk.c
```

10. automatic parallelization (Microsoft Visual Studio cl)

```
cl /GL /Gw /Ox /Qpar /Qpar-report:2 /Qvec-report:2
/Femat_mult_ijk_auto_cl.exe mat_mult_ijk.c
```

11. automatic parallelization (Oracle Developer Studio cc)

```
cc -std=c11 -m64 -fast -fd -v -xautopar -xreduction -xloopinfo
-o mat_mult_ijk_auto_cc mat_mult_ijk.c
```

12. automatic parallelization (Portland Group pgcc)

```
pgcc -c11 -m64 -fast -Minfo -Mconcur=allcores -o mat_mult_ijk_auto_pgcc
mat_mult_ijk.c
```

13. parallelization with OpenMP (GNU gcc)

```
gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-
prototypes -fopenmp -o mat_mult_OpenMP_ijk_gcc mat_mult_OpenMP_ijk.c
```

14. parallelization with OpenMP (Intel icc)

```
icc -std=c11 -m64 -fast -prec_div -Wall -Wcheck -qopenmp
-o mat_mult_OpenMP_ijk_icc mat_mult_OpenMP_ijk.c
```

15. parallelization with OpenMP (LLVM clang)

```
clang -std=c11 -m64 -O2 -fopenmp -o mat_mult_OpenMP_ijk_clang
mat_mult_OpenMP_ijk.c
```

16. parallelization with OpenMP (Microsoft Visual Studio cl)

```
cl /GL /Gw /Ox /openmp /Femat_mult_OpenMP_ijk_cl.exe mat_mult_OpenMP_ijk.c
```

17. parallelization with OpenMP (Oracle Developer Studio cc)

```
cc -std=c11 -m64 -fast -fd -v -xopenmp -xloopinfo -xvpara
-o mat_mult_OpenMP_ijk_cc mat_mult_OpenMP_ijk.c
```

18. parallelization with OpenMP (Portland Group pgcc)

```
pgcc -c11 -m64 -fast -Minfo -mp -o mat_mult_OpenMP_ijk_pgcc
mat_mult_OpenMP_ijk.c
```

(The default is **one** thread so that you must use OMP_NUM_THREADS to get a parallel execution.)

19. parallelization with MPI (GNU gcc)

(Use “mpicc -showme” (Open MPI) or “mpicc -show” (MPICH) to learn, which compiler and compiler options will be used.)

```
mpicc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes
-Wmissing-prototypes -o mat_mult_mpi_ijk_gcc mat_mult_mpi_ijk.c
```

20. parallelization with MPI (Oracle Developer Studio cc)

(Use “mpicc -showme” (Open MPI) or “mpicc -show” (MPICH) to learn, which compiler and compiler options will be used.)

```
mpicc -std=c11 -m64 -fast -fd -v -o mat_mult_mpi_ijk_cc mat_mult_mpi_ijk.c
```

21. parallelization with MPI (Intel Parallel Studio)

(Use “mpicc -show”, “mpigcc -show”, and “mpiicc -show” to learn, which compiler and compiler options will be used. Intel’s MPI is based on MPICH. It is necessary that you put a comment character in front of all lines starting with “set MPI = “ in file “$HOME/.cshrc”, if you want to use the MPI package from *Intel Parallel Studio* and that you logout and login again to deactivate any other MPI package.)

```
loki mat_mult_mpi 115 mpicc -show
gcc -I/usr/local/intel_xe_2017/...

loki mat_mult_mpi 116 mpigcc -show
gcc -I/usr/local/intel_xe_2017/...
```

```
loki mat_mult_mpi 117 mpiicc -show
icc -I/usr/local/intel_xe_2017/...

mpicc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes
-Wmissing-prototypes -o mat_mult_mpi_ijk_icc mat_mult_mpi_ijk.c
```

22. parallelization with CUDA (NVIDIA nvcc)

```
nvcc -arch=sm_50 -o dot_prod_CUDA_nvcc dot_prod_CUDA.cu
```

23. parallelization with CUDA (LLVM clang)

```
clang -m64 -O2 -L/usr/local/cuda/lib64 -o dot_prod_CUDA_clang
dot_prod_CUDA.cu -lcudart -lm
```

("-std=c11" isn't allowed if you want to compile CUDA programs, because it would result in an error message "invalid argument '-std=c11' not allowed with 'CUDA'". You can also add "--cuda-gpu-arch=sm_50 -Xcuda-ptxas –v".)

24. parallelization with OpenCL (GNU gcc, Linux)

```
gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-
prototypes -o dot_prod_OpenCL_gcc dot_prod_OpenCL.c errorCodes.c -lOpenCL
```

25. parallelization with OpenCL (GNU gcc, MacOS X)

```
gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-
prototypes -o dot_prod_OpenCL dot_prod_OpenCL.c errorCodes.c -framework
OpenCL
```

26. parallelization with OpenCL (Intel icc)

```
icc -std=c11 -m64 -O3 -prec_div -Wall -Wcheck -o dot_prod_OpenCL_icc
dot_prod_OpenCL.c errorCodes.c -lOpenCL
```

27. parallelization with OpenCL (LLVM clang)

```
clang -std=c11 -m64 -O2 -o dot_prod_OpenCL_clang dot_prod_OpenCL.c
errorCodes.c -lOpenCL -lm
```

28. parallelization with OpenCL (Microsoft Visual Studio cl)

```
cl /GL /Gw /Ox /Fedot_prod_OpenCL_cl.exe cl dot_prod_OpenCL.c errorCodes.c
OpenCL.lib
```

29. parallelization with OpenCL (NVIDIA nvcc)

```
nvcc -arch=sm_50 -o dot_prod_OpenCL_nvcc dot_prod_OpenCL.c errorCodes.c
-lOpenCL
```

30. parallelization with OpenCL (Oracle Developer Studio cc)

```
cc -std=c11 -m64 -fast -fd -v -o dot_prod_OpenCL_cc dot_prod_OpenCL.c
errorCodes.c -lOpenCL -lm
```

31. parallelization with OpenCL (Portland Group pgcc)

```
pgcc -c11 -m64 -fast -o dot_prod_OpenCL_pgcc dot_prod_OpenCL.c
errorCodes.c -lOpenCL
```

32. parallelization with OpenACC (GNU gcc)

```
gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-
prototypes -fopenacc -o dot_prod_OpenACC_gcc dot_prod_OpenACC.c
```

33. parallelization with OpenACC (Portland Group pgcc)

```
pgcc -c11 -m64 -fast -Minfo -acc -ta=nvidia -o dot_prod_OpenACC_pgcc
dot_prod_OpenACC.c
```

34. parallelization with OpenMP for accelerators (GNU gcc)

```
gcc -std=c11 -m64 -O3 -pedantic -Wall -Wstrict-prototypes -Wmissing-
prototypes -fopenmp -o dot_prod_accelerator_OpenMP_gcc
dot_prod_accelerator_OpenMP.c
```

(You can choose a default device before you run the program, e.g.,
setenv OMP_DEFAULT_DEVICE 0 (CPU)
setenv OMP_DEFAULT_DEVICE 1 (GPU))