Numerical optimisation and parallelisation

How to speed up your code

PSG group meeting

Tobias Wittwer

November 17, 2008

Delft Institute of Earth Observation and Space Systems

TU Delft

Technische Universiteit Delft

Motivation

Why numerical issues are important

- Our research is based on the results of computations
- Faster programs allow you to
  - Generate results quicker
  - Test different ideas in less time
  - Make more efficient use of your time
- Faster programs can be achieved by
  - Optimisation (increasing the speed of the code)
  - Parallelisation (using more computing power)

Overview

Contents of the presentation

Timing and profiling

Identifying computationally intensive program parts

- Only programs parts that require a lot of runtime need to be optimised
- `omp_get_wtime()` calls can be placed to identify slow program parts
- Profiling yields number of routine calls and time spent for executing routines, shows memory access problems
- Profiling usually requires unoptimised compilation and may thus lead to inaccurate numbers

Numerical optimisation

Some basic guidelines

- Compile with optimisations turned on (`-O3 -xP` on Intel systems, `-O3 -xW` on Cleopatra)
- Avoid *`if`* in loops
- Compute constant expressions only once and store them
- Expensive operators are:
  - trigonometric functions
  - square roots
  - exponentiation
  - division

Numerical optimisation

Minimise memory access

$$\begin{tikzcd}
\boxed{\text{register}} \arrow[r, leftrightarrow] & \boxed{\text{L1 cache}} \arrow[r, leftrightarrow] & \boxed{\text{L2 cache}} \arrow[r, leftrightarrow] & \boxed{\text{RAM}} \arrow[r, leftrightarrow] & \boxed{\text{disk}}
\end{tikzcd}$$

memory:	Bytes	KBytes	MBytes	GBytes	TBytes
cycles:	1	10	100	1000	10.000.000

- Memory access is a bottleneck
- Store even results of small computations in scalars when used repeatedly in a loop
- Computing something can be faster than retrieving it from memory

Numerical optimisation

Series computations

$$\beta(x) = \sum_l \frac{2l+1}{4\pi} \left(\frac{R}{r}\right)^l \alpha(x,l)$$

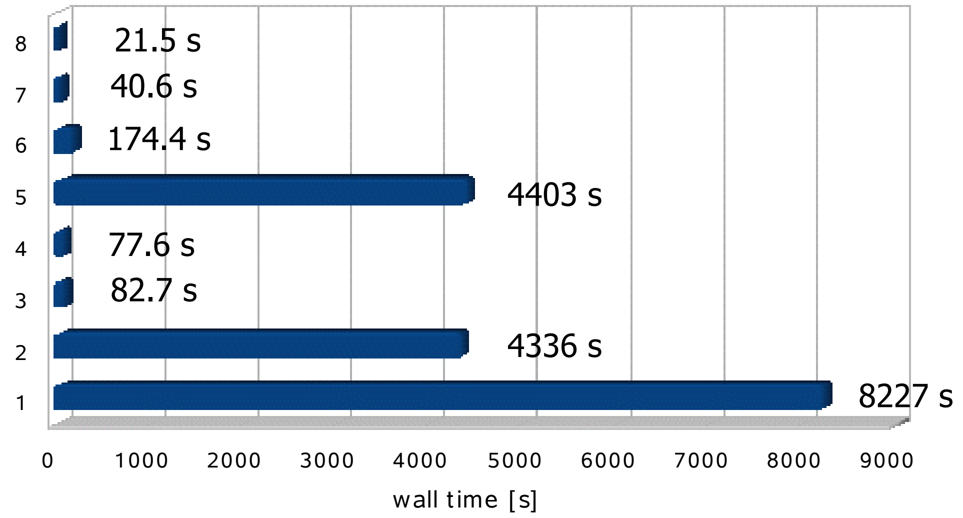
- Constant expressions should be computed only once
- Exponentiation is expensive, can be computed recursively:

$$\left(\frac{R}{r}\right)^l = \left(\frac{R}{r}\right)^{l-1} \left(\frac{R}{r}\right)$$

TU Delft

Matrix-matrix multiplication

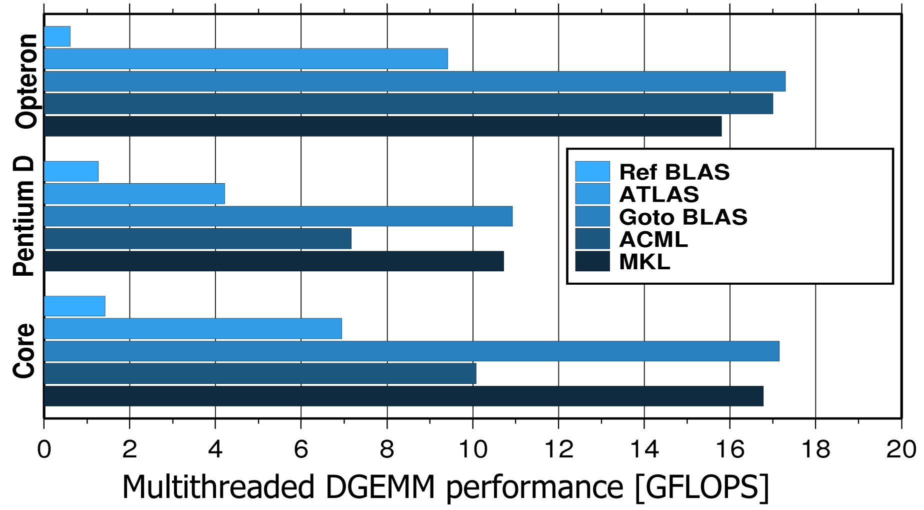
Runtime comparison

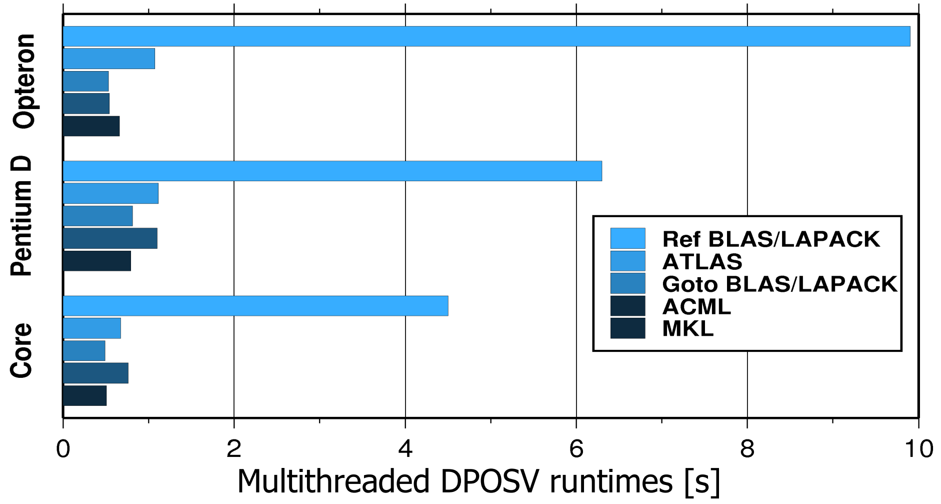


8 – DGEMM MKL 2 threads
7 – DGEMM MKL 1 thread
6 – DGEMM reference BLAS
5 – matmul()
4 – triple loop -O3 -xP -parallel
3 – triple loop -O3 -xP
2 – triple loop -O2
1 – triple loop -O0

- dual-core Pentium D, Intel FC 10.1
- vectorisation is fast (`-O3 -xP`)
- `matmul()` is very slow!
- reference BLAS is slow
- MKL is fast and parallelised
- BLAS can use `DSYRK` for N=A'A (50% faster)

Choosing the right BLAS/LAPACK lib





- Reference BLAS is slow!
- Goto BLAS or vendor-specific libraries should be used – optimised and parallelised
- LAPACK performance is governed by BLAS performance

TUDelft

Why packed storage is bad

Packed storage LAPACK routines are slow

- Cholesky factorisation of 10201x10201 matrix
- Cleopatra, 4 threads
- Goto BLAS + LAPACK
- Unpacked: DPOTRF, 26.7 s
- Packed: DPPTRF, 631 s
- Packed storage offers less than 50% memory benefit, but is more than 20 times slower
- Avoid packed storage when runtime is important!

Parallelisation

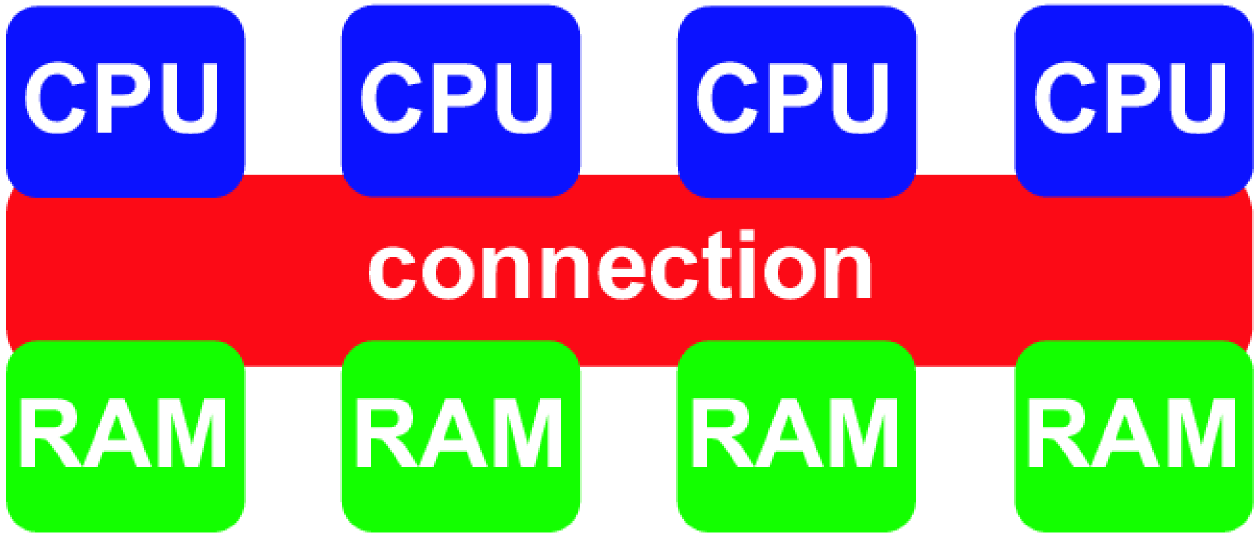
Why parallelisation is necessary

- Gigahertz-race has stopped
- Further developments aim at increasing the number of “cores” per CPU
- Parallel programming is required to make use of these parallel computer architectures
- Good parallel programming almost eliminates memory and computational restrictions

TUDelft

Parallel computer architectures

Shared memory

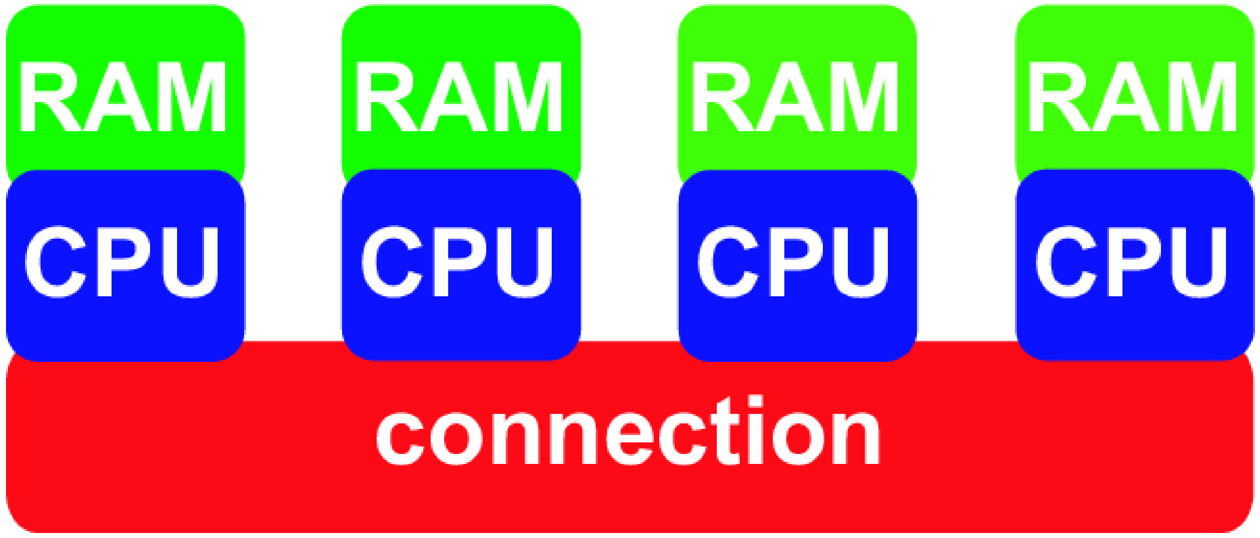


- Easy to program, fast
- Expensive, limited maximum size

TU Delft

Parallel computer architectures

Distributed memory

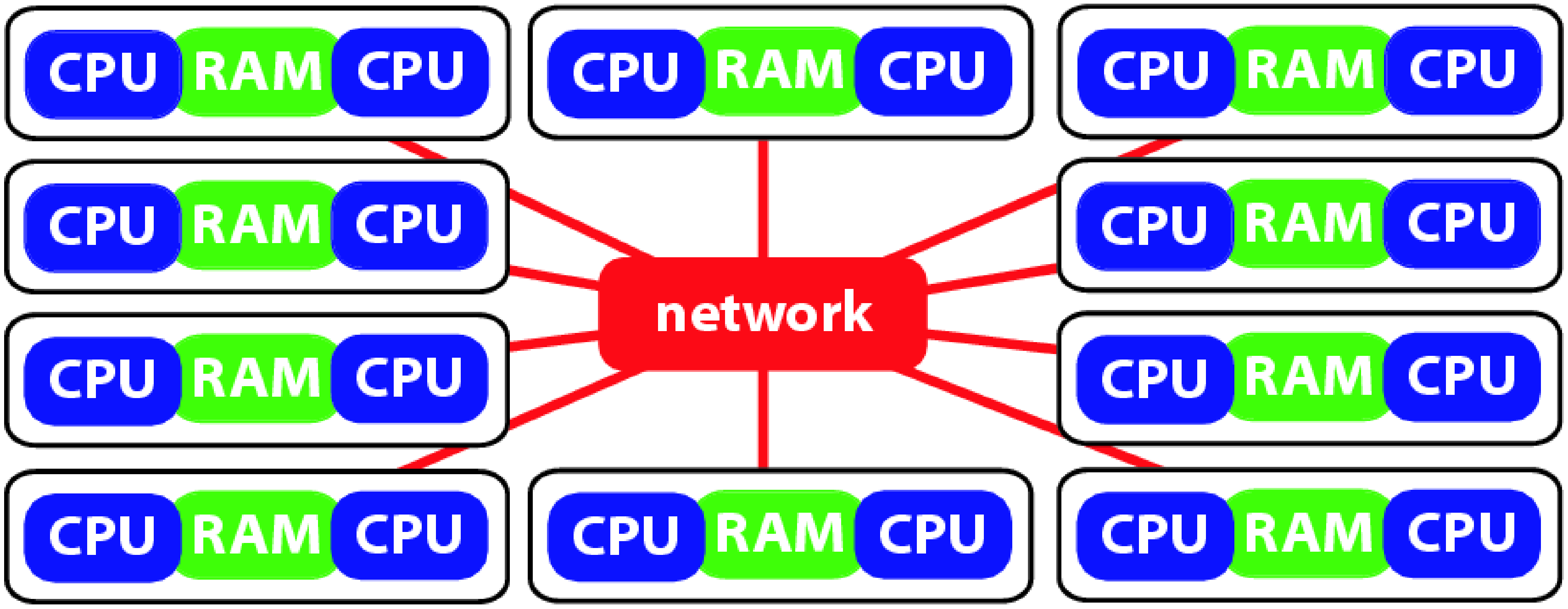


- Almost unlimited maximum size
- More difficult to program, slow if lots of communication is required

TU Delft

Parallel computer architectures

Clusters



- Hybrid architecture, can be built from cheap components
- Similar benefits and limitations as distributed mem systems

Shared memory parallelisation

OpenMP

- OpenMP is used to parallelise loops
- Needs to be supported by the compiler
- Very easy to use

```
!$omp parallel do
do i=1,n
  A(i) = f(i)
end do
!$omp end parallel do
```

- Load balancing is done by operating system
- Memory-intensive programs might not benefit

TUDelft

Distributed memory parallelisation

MPI

- MPI is used for explicitly exchanging data between processes

```
do i=myrank+1,n,nprocs
  A(i) = f(i)
end do
call mpi_allreduce(A,...)
```

- No automatic load balancing
- Performance is dependent on communications network
- Communication should be kept to a minimum

TUDelft

Distributed memory parallelisation

ScaLAPACK

- ACML, MKL, Goto BLAS & LAPACK routines are parallelised for shared memory only
- ScaLAPACK contains efficient distributed-memory routines of BLAS and LAPACK functionality
- Matrices are distributed among processes, full system memory can be used
- Uses MPI for communication:
  - Performance is dependent on network
  - Communication should be kept to a minimum
- Relatively complicated to use

TUDelft

Computing resources

Cleopatra

- 33 nodes
- 4 Opteron 280 nodes, 2.4 GHz
- 32 nodes with 8 GB RAM, 1 node with 16 GB RAM
- 272 GB total RAM
- Infiniband network
- 633 GFLOPS peak performance
- exclusively available to our group

TUDelft

Computing resources

Huygens 2

- 104 nodes
- 32 Power 6 cores, 4.7 GHz
- 83 nodes with 128 GB RAM, 18 nodes with 256 GB RAM
- 15.6 TB total RAM
- Infiniband network
- 60 TFLOPS peak performance
- limited CPU time budget!

TU Delft

Conclusions

Things to remember

- Compile with `-O3 -Xp` / `-O3 -Xw`
- Memory access is slow
- Small optimisations can speed up your code significantly
- Use BLAS and LAPACK routines for linear algebra
- Use ACML, MKL or Goto implementations
- Don't use packed storage for runtime-critical code
- Don't use Matlab for runtime-critical code
- Parallelise when necessary