Computational Astrophysics

parallel programming

Peter Hauschildt
yeti@hs.uni-hamburg.de

Hamburger Sternwarte
Gojenbergsweg 112
21029 Hamburg

24. Oktober 2018

Topics

- vectorization
- parallelization models
- shared memory model (openMP)
- distributed memory model (MPI)
- practical example: `PHOENIX`

acknowledgment

- 'The original MPI training materials were developed under the Joint Information Systems Committee (JISC) New Technologies Initiative by the Training and Education Centre at Edinburgh Parallel Computing Centre (EPCC-TEC), University of Edinburgh, United Kingdom.'
- in addition, material from the Konrad-Zuse-Zentrum (Berlin) was used (W. Baumann, H. Stüben)

introduction

- in many cases, performance is not critical
- however, if CPU/wallclock times are long, performance can be critical
- first step: standard optimization practices
- many of those are done by the compiler
- if it is instructed to do so!
- $\rightarrow$ *RTFM*

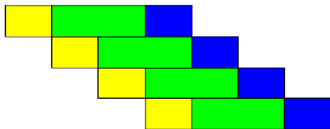
introduction

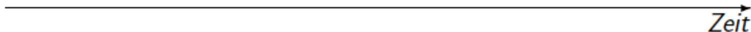
- good coding practices help
- sometimes this is not enough
- some problems are just too large
- $\rightarrow$ use more advanced techniques
- classical approach:
- *vector processing*
- introduced by Seymore Cray, early 80's.

vectorization

Skalare Verarbeitung = serielle Vearbeitung:

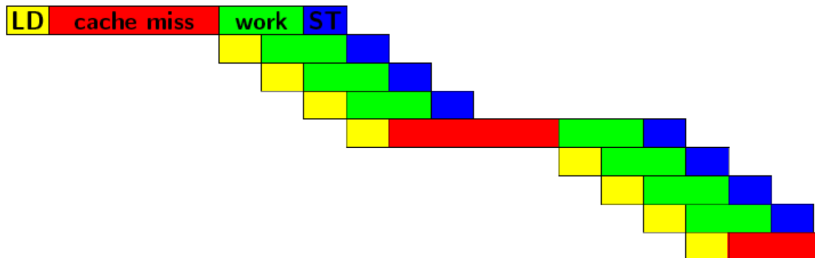
Vektorverarbeitung = Fließbandverarbeitung:





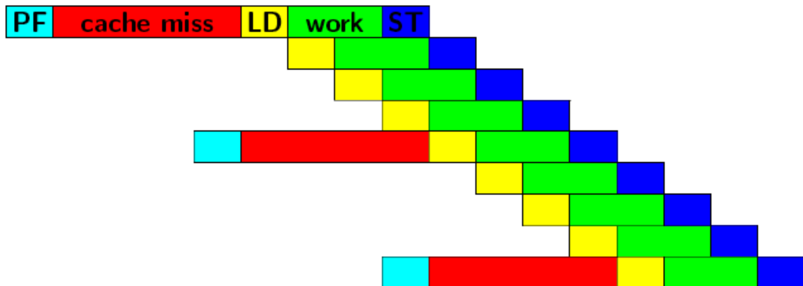
vectorization

- also useful in modern CPUs via prefetch:
- standard, no prefetch



vectorization

- with prefetch



vectorization examples

- vectorizable loops:

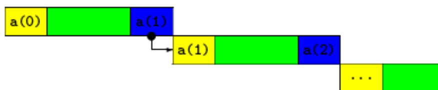
```
do i = 1, N
   a(i) = a(i) + 1
enddo
```

```
do i = 1, N
   a(i) = a(i + 1) + 1
enddo
```

vectorization examples

- data dependency preventing vectorization:

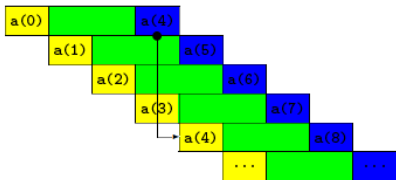
```
do i = 1, N
   a(i) = a(i - 1) + 1
enddo
```



vectorization examples

- data dependency, safe vector length

```
do i = 4, N
   a(i) = a(i - 4) + 1
enddo
```



vectorization

- compilers can *vectorize* loops
- for this to work, they have to be written 'correctly' by the user
- often compiler directives are used to aid the compiler to avoid data dependencies etc.
- but there are always loops that cannot be vectorized
- speed-ups can be factors 2-25

vectorization

- typical loop types:

1.	*full vector*	`a(i) = b(i)`
2.	*gather*	`a(i) = b(j(i))`
3.	*scatter*	`a(j(i)) = b(i)`
4.	*atomic update*	`a(j(i)) = a(j(i)) + b(i)`
5.	*reduction*	`s = s + a(i)`
6.	*„Jacobi“*	`a(i) = (b(i - 1) + b(i + 1)) / 2.0`
7.	*„Gauß-Seidel“*	`a(i) = (a(i - 1) + a(i + 1)) / 2.0`

vectorization

- 1, 2, 5, 6 are vectorizable
- 3, 4 are vectorizable with directives
- 7 is vectorizable with checkerboard methods
- vectorization is good, but sometimes not successful

parallelization

- vector processors hit performance limits quickly
- pipelines need to be filled (latencies)
- produce at best one result per cycle
- $\rightarrow$ processor cycle and memory speed limit performance
- for further improvements, parallelization is used

parallelization

- there are two main parallelization concepts:
  1. *single program, multiple data* $\rightarrow$ *SIMD*
  2. *multiple program, multiple data* $\rightarrow$ *MIMD*
- each of these is used in practical applications
- the SIMD model is considered easier to use
- (I don't think so)

parallelization

- SIMD parallel programs are used on *shared memory* parallel machines
- here, several CPUs share a common memory subsystem
- *shared memory processing* or *symmetric multiprocessing* (SMP) machines
- require complex hardware to keep cache coherency etc
- $\rightarrow$ expensive machines (especially if > 2 CPUs)

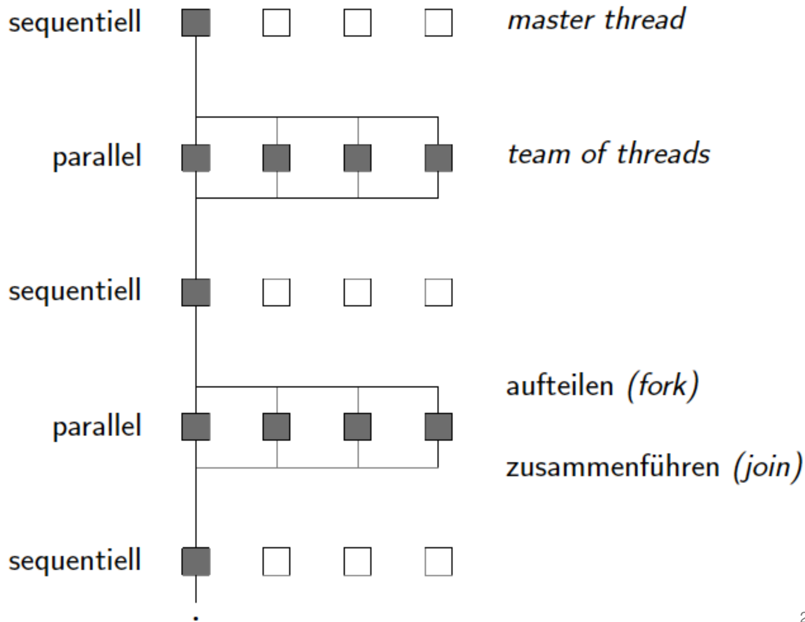
parallelization

- how to program SIMD code on a SMP?
- typically done on the 'loop-level'
- one solution:
- HPF $\rightarrow$ *high performance fortran*
- a set of compiler directives to allow parallel execution of loops
- not very flexible, but portable

openMP

- more general approach: *openMP*
- set of directives for fortran and C
- allow more complicated parallelization (not only loop level)
- assumes a SMP machine!
- basic concept:

openMP



openMP

- examples:
- simple parallel loop:

```
!$omp parallel do
do i = 1, N
    a(i) = b(i)
enddo
```

- atomic update:

```
!$omp parallel do
do i = 1, N
    !$omp atomic
    a(j(i)) = a(j(i)) + b(i)
enddo
```

openMP

- reduction:

```
s = 0.0
!$omp parallel do reduction(+: s)
do i = 1, N
   s = s + a(i)
enddo
```

- Jacobi:

```
!$omp parallel do
do i = 1, N
   a(i) = (b(i - 1) + b(i + 1)) / 2.0
enddo
```

openMP

- parallel routines:

```
!$omp parallel
call b                static extent
call c
!$omp end parallel
```

- thread private variables:

```
!$omp parallel do private(y)
do i = 1, N                          ! i ist private
   y = f(x(i))                       ! y ist private
   a(i) = a(i) + c + y               ! c ist shared
   b(i) = b(i) + d + y**2            ! d ist shared
enddo
```

openMP

- there are many more directives
- the standard is described at:
- `http://www.openmp.org/`
- openMP works well for some programs

openMP

- but typical problems are
  - poor scalability (depending on loop length)
  - implicit barriers causing slowdown
  - stupid and buggy compiler support
  - not much user control over parallelization and communication
- $\rightarrow$ openMP is good for some programs but not useful for many codes

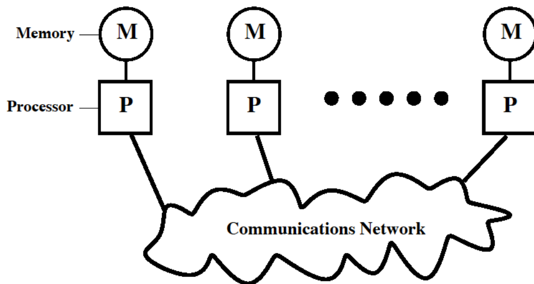
MPI

- SMP hardware is expensive
- cheaper to use many simple machines for parallel codes
- how to write parallel codes on such machines?
- how to write more general (MIMD) programs?
- how to use aggregate memory of many machines?
- presently, this type of parallelization cannot be done automatically
- or through directives

MPI

- solution: use explicit communication between different machines
- this works also on SMP machines!
- the communication is handled through (public domain) libraries
- most frequently used (today):
- *message passing interface* MPI
- basic programming concept:

MPI



MPI

- each processor in a MPI program runs a different copy of the code
- written in a conventional sequential language.
- all variables are private.
- communicate via special subroutine calls.
- messages are packets of data moving between MPI tasks

MPI

- message passing system has to be told the following information:
    - sending processor
    - source location
    - data type
    - data length
    - receiving processor(s)
    - destination location
    - destination size

MPI

- important: receiving process is capable of dealing with messages it is sent
- Point-to-Point Communication:
  - one process sends a message to another
  - simplest form of message passing
  - different types of point-to-point communication

MPI

- Synchronous Sends:
- provide information about the completion of the message
- Asynchronous Sends
- only know when the message has left
- Blocking Operations:
- only return from the subroutine call when the operation has completed

MPI

- Non-Blocking Operations
- return straight away and allow the MPI task to continue to perform other work
- At some later time the MPI task can test or wait for the completion of the non-blocking operation

MPI

- Collective communications:
- higher level routines involving several MPI processes at a time
- Barriers:
- synchronize processes
- Broadcast:
- one-to-many communication

MPI

- Reduction Operations:
- combine data from several processes to produce a single result
- MPI provides facilities ('communicators' to address groups of MPI tasks
- each MPI process has a 'rank' to identify it
- MPI also works across heterogeneous clusters!
- MPI Pt-2-Pt operations:

MPI

OPERATION	MPI CALL
Standard send	MPI_SEND
Synchronous send	MPI_SSEND
Buffered send	MPI_BSEND
Ready send	MPI_RSEND
Receive	MPI_RECV

MPI

NON-BLOCKING OPERATION	MPI CALL
Standard send	`MPI_ISEND`
Synchronous send	`MPI_ISSEND`
Buffered send	`MPI_IBSEND`
Ready send	`MPI_IRSEND`
Receive	`MPI_IRECV`

MPI

- MPI codes require user design
- steep learning curve...
- this sounds harder than it is
- it is actually easier than getting SMP programs to work efficiently
- it is important to realize that the machines only communicate if told to!
- there are many more sources of error in parallel programs
- like the dreaded deadlock . . .

Example: `PHOENIX` code

- general-purpose stellar atmosphere code
- implements detailed micro-physics
- in development for $\approx$ 15 years
- portable
- about 1.8M lines of Fortran, C, C++ code
- applied successfully to a large variety of problems

PHOENIX code

- parallelization to allow larger and more detailed simulations in reasonable timeframe.
- independent physical & logical program modules
  $\rightarrow$ allows task parallelism
  $\rightarrow$ plus data parallelism within each module
- problem: very different types of simulations
- $\rightarrow$ require different algorithms

(serial) CPU time

- small for each individual point on the wavelength grid: $\approx 10 \ldots 100$ msec
- number of wavelength points for radiative transfer: 30,000-300,000 (can be $> 10^6$)
- $\rightarrow$ up to 30,000 sec to “sweep” once through all wavelength points
- typically, $\approx$ 10 iterations (sweeps) are required to obtain an equilibrium model
- $\rightarrow \approx 3.5$ CPU days
- there are, literally, 10000’s of models in a typical grid

…

Solution through parallelization

- large number of simulations
- complex code (verification on several different architectures)
- need to be able to run *efficiently* on different parallel supercomputers
- $\rightarrow$ Fortran & `MPI`
- available on all major platforms
- public domain implementations: `MPICH, LAM`

Solution through parallelization

- memory issues
  - `MPI` available on distributed memory systems
  - large aggregate memory of distributed memory machines
  - allows reduction of memory requirements per PE
  - $\rightarrow$ larger model calculations possible!

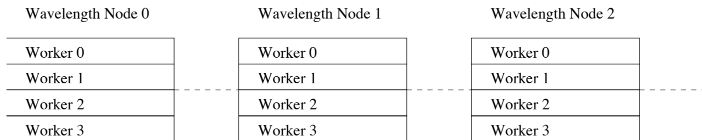
Solution through parallelization

- scalability issues
  - allows more efficient usage of multiple CPUs
  - reduces wall-clock time for typical simulations
  - depends very often on type of models:
    some simulations (stars) allow algorithms that scale very well, but some simulations (novae, SNe) do not
  - $\rightarrow$ implement several algorithms that can be selected at run-time to obtain “best” overall performance while making simulations feasible!

Parallelizing the wavelength loop

- "longest" loop in the whole code: number of wavelength points!
- $\rightarrow$ ideal for parallelization
- works extremely well for *static* configurations: each wavelength point can be done in parallel with *no* communication until each PE has completed its sweep.
- does *not* reduce memory requirements per PE
  $\rightarrow$ combine with other task/data parallel algorithms
  $\rightarrow$ concept of "wavelength clusters" with a set of "worker PEs" each

Design



Parallelizing the wavelength loop

- expanding atmospheres: radiative transfer is *initial value problem* in wavelength
- wavelength point l depends on previous point $l - 1$
- $\rightarrow$ use a “pipeline” approach to parallelization
  $\rightarrow$ cluster working on point $l - 1$ sends data to cluster working on l

Parallelizing the wavelength loop

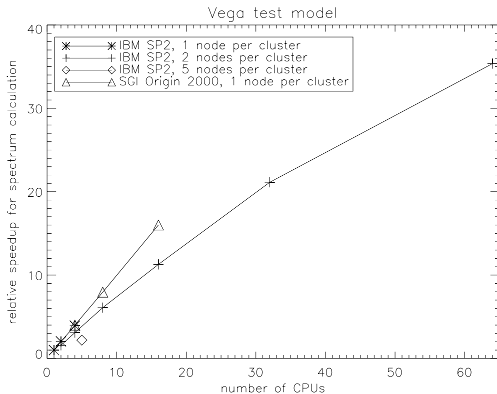
- problem separates into pre– and post–processing phases:

```
for i := 1 to NUMWAVELENGTHS
   pre_processing: {
     ...
     atomicLineOpacity(...)
     molecularLineOpacity(...)
     nlteOpacity(...)
   }
   radiativeTransfer(...)
   post_processing: {
     ...
     nlteUpdateRates(...)
   }
end
```

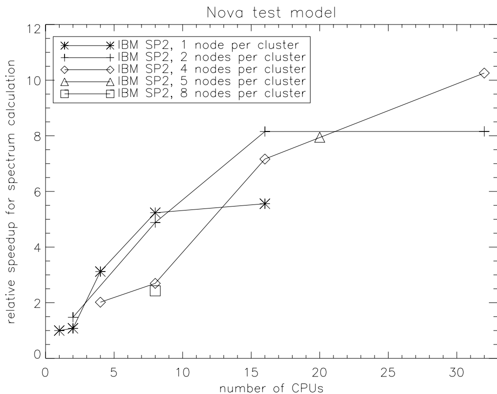
Parallelizing the wavelength loop

- properties similar to vector pipeline
- limited scalability
- combination of clusters and workers can be used to increase performance on any given architecture
- performance depends on type of model:
- static $\rightarrow$ nearly perfect scaling
- moving $\rightarrow$ RT causes stalls

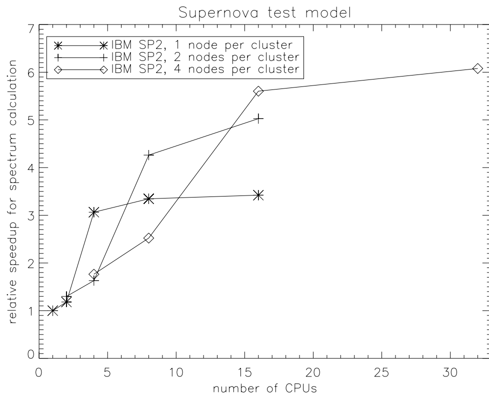
static model



nova model



supernova model



Conclusions

- parallelization of `PHOENIX` allows physically more detailed models
- decrease in wall-clock time per model is substantial for many types of simulations
- coding effort to implement `MPI` calls relatively small (about 33000 lines or 2%)
- logic for algorithm selection and load balancing fairly complex
- parallel version of `PHOENIX` is regularly used in production
- depending on simulation type we use between 4 and 2.5M PEs (single core)