Introduction into Hardware and parallel concepts

Gundolf Haase

Institute for Mathematics and Scientific Computing
University of Graz, Austria

Isfahan, Jan 2019

What do you have to expect?

Tutorial with practical work

I) Parallel concepts; Hardware; Relations inbetween; The example environment

II) PDE; Finite Element discretization; System of equations; (simple) iterative solver; parallelization concept

III) Classical shared memory and distributed parallelization parallelization (OpenMP and MPI)

IV) Accelerator programming for NVIDIA GPU; practical work (OpenACC and CUDA)

V) Distributed computing with multiple GPUs; practical work

Parallel concepts

Classification by memory access

- Distributed memory access
  - cluster computing; multi-core computing
  - MPI (Message Passing Interface)
- Shared memory access
  - multi-core computing, many-core computing
  - OpenXXX, CUDA, OpenCL
  - distributed shared memory on compute clusters available.
- Faked shared memory access
  - distributed shared memory on compute clusters available (hardware!!).
  - PGAS (partitioned global address space)

UMA: uniform memory access
NUMA: non-uniform memory access
ccNUMA: cache coherent NUMA
hUMA: heterogeneous UMA (by AMD)

bandwidth: byte per second in a data transfer from/to memory $(\mathcal{O}(\frac{1}{t_{\text{bandwidth}}}))$
latency: time until data transfer starts t_{latency}
Transferring n Byte: $t(n) = t_{\text{latency}} + n * t_{\text{bandwidth}}$

Memory hierarchies

- Normal DRAM (Dynamic Random Access Memory) stores a bit in a capacitor
  - needs only a few transistors $\Longrightarrow$ small area on chip, cheap
  - large amount of memory
  - needs refreshment cycles $\Longrightarrow$ slow access
- Cache SRAM (Static Random Access Memory) stores a bit in a flip-flop circuit
  - needs more transistors $\Longrightarrow$ larger area on chip, expensive
  - small amount of memory
  - no refreshment cycles $\Longrightarrow$ fast access
- Therefore, DRAM is combined with a hierarchy of smaller but faster caches.

Non **U**niform **M**emory **A**ccess (wrt. latency and bandwidth)

- CPU:
  Register – L1 – L2 – L3-cache – memory – remote memory
- GPU:
  Register – shared/L1 – L2 – GPU memory – CPU memory

Classification by streams [Flynn, 1966]

Data stream vs. Instruction stream

Instruction Stream			
Single	Multiple		
SISD	**MISD**	Single	Data Stream
SIMD	**MIMD**	Multiple	

Table: Flynn's taxonomy

SISD (Single Instruction Single Data) is the classical sequential von-Neumann computer.

MISD (Multiple Instruction Single Data) can be found in the pipelining of instruction in modern processors and in flight control computers.

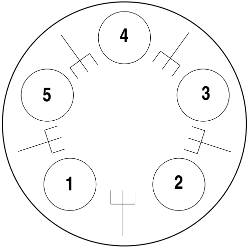
Our focus will be on **MIMD** and **SIMD**.

MIMD - Multiple Instructions Multiple Data

- Each process (and its instructions) access data on different ressources (i.e., distributed memory)
- Often as SPMD (Single Programm Multiple Data)
- explicit access to ressources of other processes via communication.
- $\Longrightarrow$ dead locks might block the whole code
- MPI (Message Passing Interface)

Dead lock: Dinner for five [Dijkstras 1971]

Dead lock: *Processes have to wait for an event that has to be performed by one of the waiting processs.*



5 philosophers (P) with 5 forks (R).
Each P needs two forks (R) for eating:

1. Each philospher takes the the right fork and waits for the left fork. $\Longrightarrow$ Dead lock for all (starving with one fork in their hand)
2. Wait until both forks are available, eat and release them afterwards $\Longrightarrow$ Dead lock for one (P_1, P_3) eat alternating with (P_5, P_2) and P_4 starves

A dead lock for all is obvious but a dead lock for one might be very subtle to find.

SIMD - Single Instruction Multiple Data

- Each thread (and its instructions) accesses data on shared ressources (e.g., shared memory)
- implicit access to ressources of other threads.
- $\Longrightarrow$ data races result in unpredictable (incorrect) results
- OpenXXX, CUDA, OpenCL
- A SIMT (Single Instruction Multiple Threads) per warp on GPUs available.
  - 1 instruction pointer per b threads in one warp
  - all b threads have to wait for slowest thread (alternatives, while-loops)

Data Race

Uncoordinated manipulation of shared ressources.

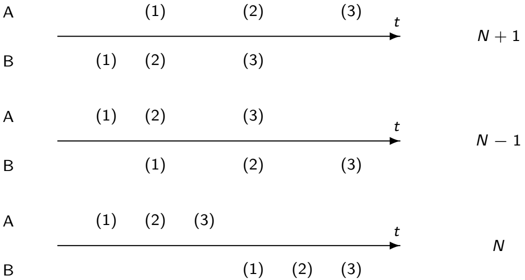
thread A: $N := N+1$	clock.	thread B: $N := N-1$
`load` N	(1)	`load` N
`inc` N	(2)	`dec` N
`store` N	(3)	`store` N

The value of N is not predictable, it depends on the execution speed of threads A and B

Case a)

Case b)

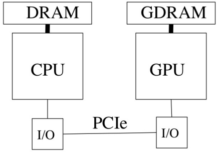
Case c)

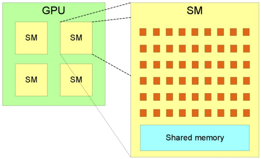


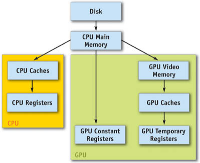
Data race: solution

- Consider the operations `load`, `inc`/`dec`, `store` as **one atomic** operation.
- This atomic operation has to be finished before another thread gets access to the ressources.
- $\Longrightarrow$ N will be locked.
- N in local cache requires ccNUMA (Hey, that value has been changed!).
- For-loops for vector operation $\underline{z} = \alpha \cdot \underline{z} + \underline{y}$ followed by $\underline{a} = \underline{z} + \underline{b}$ might require thread synchronization between loops.
(`#pragma omp barrier`)

What is new in accelerator programming?







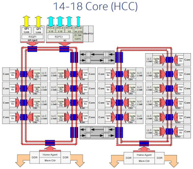
©PRACE
©NVIDIA

- 2 levels of shared memory on GPU: global for GPU + local on SM
- **Host**: Usually a CPU core
- **Device**: Accelerator device as GPU or Xeon Phi (or multi-core CPU)
- seperate memory with explicit data transfer between host and device memory
- memory(host) $>>$ memory(device)
- bandwidth(host)$<$ bandwidth(device)
- Synchronization between threads only locally on SM, not globally.
- Threads in one warp are parallel by instruction (one IP for all)

Hardware remarks

Processor types on the market (2014)

CPU
Xeon E7-8890 v2



18 cores, 1.5TB
108 GByte/sec
≈750 GFLOPS(d,peak)
AVX2 (512)
145 Watt

SIMT/SIMD (MIMD)
g++, OpenMP

Cluster on DIE;

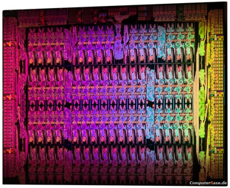
GPU
Tesla K40

15 × 192 cores, 12 GB
288 GByte/sec
1.4 TFLOPS (d,peak)
235 Watt

SIMT + MIMD
CUDA, OpenACC

incl. GPUs + OpenACC;

MIC
Xeon Phi 7120P



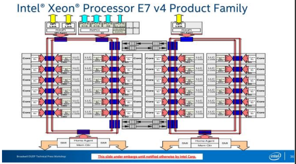
61 cores, 16 GB
352 GByte/sec
1.2 TFLOPS (d,peak)
AVX (512)
300 Watt

SIMT/SIMD (MIMD)
Intel-Compiler,
OpenMP 4.0

MIC (Many Integrated Core);

Processor types on the market (2017)

CPU
Xeon E7-8890 v4

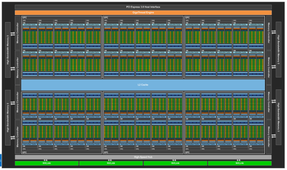

Intel® Xeon® Processor E7 v4 Product Family
QPI Link
R3QPI
Core
LLC
Home Agent
Mem Ctlr
SMI
This slide under embargo until notified otherwise by Intel Corp.

24 cores, 1.5TB
85 GByte/sec
≈844 GFLOPS(d,peak)
AVX2 (512)
165 Watt

SIMT/SIMD (MIMD)
g++, OpenMP

Cluster on DIE;

GPU
Tesla P100



56 × 64 cores, 16 GB
732 GByte/sec
4.7 TFLOPS (d,peak)
300 Watt

SIMT + MIMD
CUDA, OpenACC

incl. GPUs + OpenACC;

KNL
Xeon Phi 7290F

72 cores, 16 GB
400+ GByte/sec
3.45 TFLOPS (d,peak)
AVX512
260 Watt

SIMT/SIMD (MIMD)
Intel/GNU, OpenMP

MIC (Many Integrated Core);

Programming Models in multi-/many-core environments

- distributed memory: MPI ⊥
- vectorization: SSE, AVX ⟶ compiler support (`#pragma omp simd`)
- shared memory: OpenMP ⟶ compiler support (`#pragma omp parallel for`)
- many-core:
  - GPU-systems: CUDA, OpenCL, OpenACC (⟶ general devices)
  - general: OpenACC (`#pragma acc parallel loop`)
    commercial compiler support since spring 2012
    [Nov 13, 2011; Cray, Nvidia, PGI]
  - MIC-systems: OpenMP 4.0 (⟶ general devices) (`#pragma omp target`)
    [July 2013: AMD, Cray, Intel, IBM, NVIDIA, . . .]

Shared memory: first example

seq: Scalar product

$$s = \langle x, y \rangle = \sum_{k=0}^{N-1} x_k \cdot y_k$$

Listing 1: Scalar product

```
double scalar(int N, const double x[], const double y[])
{
    double sum = 0.0;
    for (int i=0; i<N; ++i)
    {
       sum += x[i]*y[i];
    }
    return sum;
}
```

dualhex: 2× AMD Opteron 2427, 6x 2.20GHz, 32 GB

N=250 Mill., 50 outer loops 1 core: **0.78 sec.**

shm: Scalar product – race condition

Listing 2: Scalar product with race condition

```
double scalar(int N, const double x[], const double y[])
{
   double sum = 0.0;
   int i;
#pragma omp parallel for private(i) shared(x,y,sum)
   for (i=0; i<N; ++i)
   {
      sum += x[i]*y[i];
   }
   return sum;
}
```

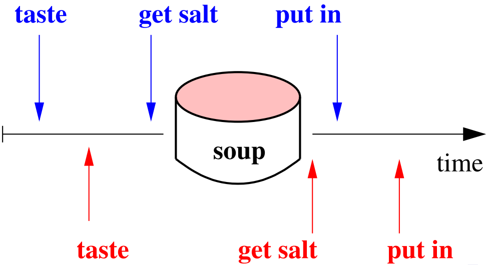
dualhex:
N=250 Mill., 50 outer loops 4 cores: **0.81 sec.**

Easy, but wrong result because of data race.

shm: data race – two cooks

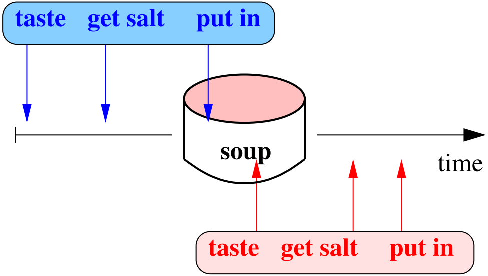
Listing 3: Scalar product with data race

```
double scalar(int N, const double x[], const double y[])
{
   double sum = 0.0;
   int i;
#pragma omp parallel for private(i) shared(x,y,sum)
   for (i=0; i<N; ++i)
   {
      sum += x[i]*y[i];
   }
   return sum;
}
```

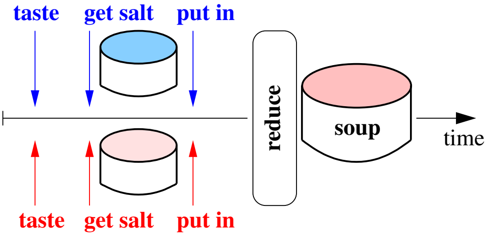


shm: two cooks – atomic vs. reduce

atomic:



reduce:



shm: Scalar product – atomic

Listing 4: Scalar product with atomic pragma

```
double scalar(int N, const double x[], const double y[])
{
   double sum = 0.0;
#pragma omp parallel for shared(x,y,sum)
   for (int i=0; i<N; ++i)
   {
#pragma omp atomic
      sum += x[i]*y[i];
   }
   return sum;
}
```

dualhex:
N=250 Mill., 50 outer loops 4 cores: **38 sec.**

correct result but slow because of atomic operation.

shm: Scalar product – reduce

Listing 5: Scalar product with reduction

```
double scalar(int N, const double x[], const double y[])
{
   double sum = 0.0;
#pragma omp parallel for shared(x,y) reduction(+:sum)
   for (int i=0; i<N; ++i)
   {
      sum += x[i]*y[i];
   }
   return sum;
}
```

dualhex:

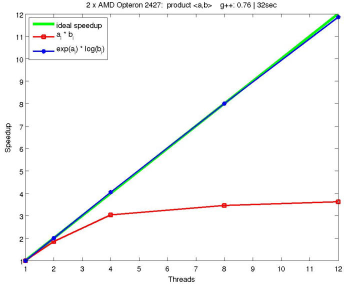
N=250 Mill., 50 outer loops

4 cores: **0.48 sec.**
(1 core: 0.78 sec.)

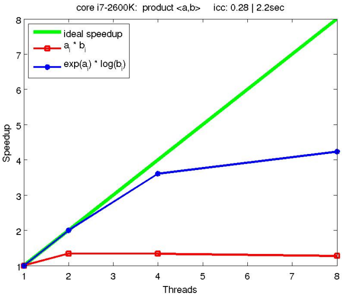
Easy, correct result.

Shared Memory: speedup

12-core Opteron



4-core SandyBridge



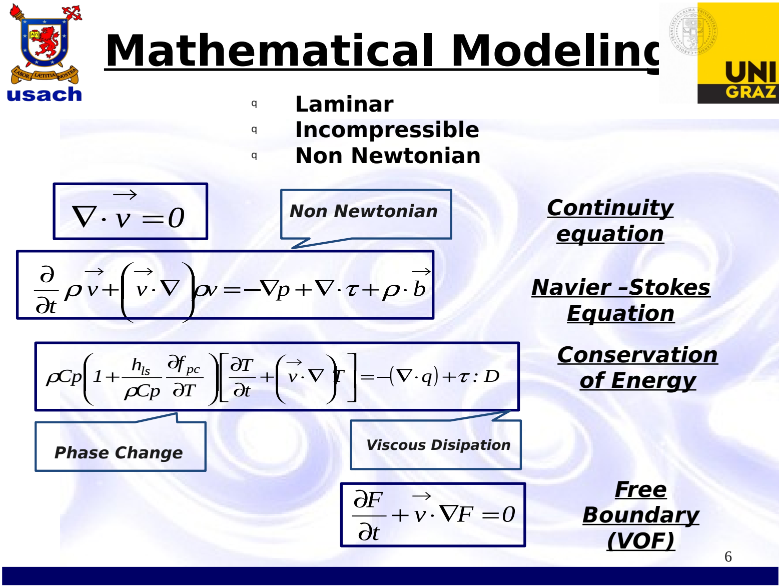
$*$ poor speedup for $\sum_{k=0}^{N-1} x_k \cdot y_k$: limited by memory bandwidth

$*$ excellent speedup for $\sum_{k=0}^{N-1} \exp(x_k) \cdot log(y_k)$:

limited by functional units

Shared memory: non-Newtonian fluid

Diego A. Vasco
[Universidad de Santiago de Chile]



Mathematical Modeling

- Laminar
- Incompressible
- Non Newtonian

$$\nabla\cdot\vec{v}=0$$

Continuity equation

Non Newtonian

$$\frac{\partial}{\partial t}\rho\vec{v}+\left(\vec{v}\cdot\nabla\right)\rho v=-\nabla p+\nabla\cdot\tau+\rho\cdot\vec{b}$$

Navier -Stokes Equation

$$\rho Cp\left(1+\frac{h_{ls}}{\rho Cp}\frac{\partial f_{pc}}{\partial T}\right)\left[\frac{\partial T}{\partial t}+\left(\vec{v}\cdot\nabla\right)T\right]=-(\nabla\cdot q)+\tau:D$$

Conservation of Energy

Phase Change

Viscous Disipation

$$\frac{\partial F}{\partial t}+\vec{v}\cdot\nabla F=0$$

Free Boundary (VOF)

usach

Mathematical Modeling

UNI GRAZ

Constitutive Equations

Normal components **Tangencial components**

Shear stress tensor

$$\tau = \begin{vmatrix} \tau_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \tau_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \tau_{zz} \end{vmatrix}$$

$$\tau_{xx} = -\eta\left[2\frac{\partial u}{\partial x}\right]$$

$$\tau_{yy} = -\eta\left[2\frac{\partial v}{\partial y}\right]$$

$$\tau_{zz} = -\eta\left[2\frac{\partial w}{\partial z}\right]$$

$$\tau_{yz} = \tau_{zy} = -\eta\left[\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right]$$

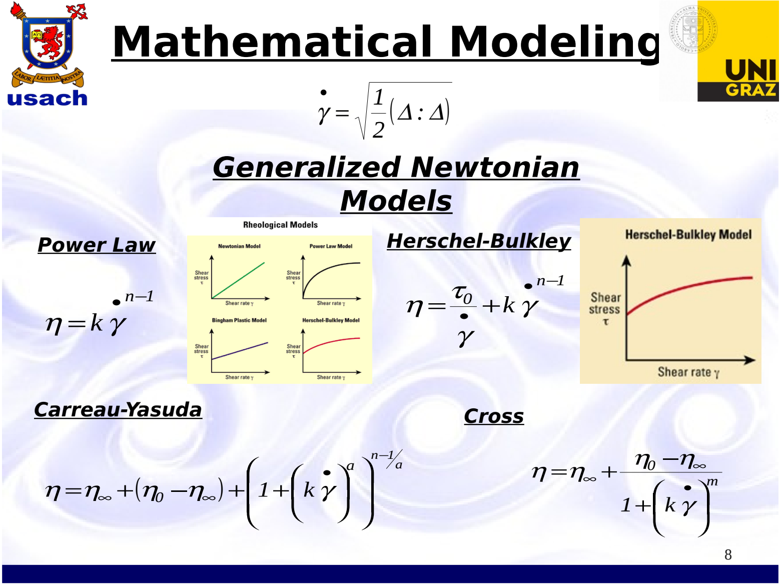
$$\tau_{xy} = \tau_{yx} = -\eta\left[\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right]$$

$$\tau_{xz} = \tau_{zx} = -\eta\left[\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right]$$

Shear rate tensor

$$\Delta_{ij} = \frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}$$

$$I_1 = (\Delta : \delta) \qquad \boxed{I_2 = (\Delta : \Delta)} \qquad I_3 = det\,\Delta$$



Mathematical Modeling

$$\dot{\gamma} = \sqrt{\frac{1}{2}(\Delta : \Delta)}$$

Generalized Newtonian Models

Power Law

$$\eta = k\,\dot{\gamma}^{\,n-1}$$

Rheological Models

Herschel-Bulkley

$$\eta = \frac{\tau_0}{\dot{\gamma}} + k\,\dot{\gamma}^{\,n-1}$$

Herschel-Bulkley Model

Carreau-Yasuda

$$\eta = \eta_\infty + (\eta_0 - \eta_\infty) + \left(1 + \left(k\,\dot{\gamma}\right)^{a}\right)^{{n-1}/{a}}$$

Cross

$$\eta = \eta_\infty + \frac{\eta_0 - \eta_\infty}{1 + \left(k\,\dot{\gamma}\right)^{m}}$$

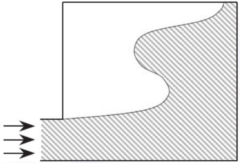
Mathematical Modelin

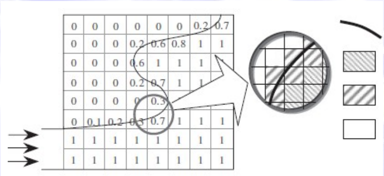
Moving Boundary

VOF (Volume of Fluid) Method

$$\frac{\partial F}{\partial t} + \vec{v} \cdot \nabla F = 0$$

$$\int_V \frac{\partial F}{\partial t} \, dV + \int_s F \, \vec{v} \cdot \hat{n} \, ds = 0$$



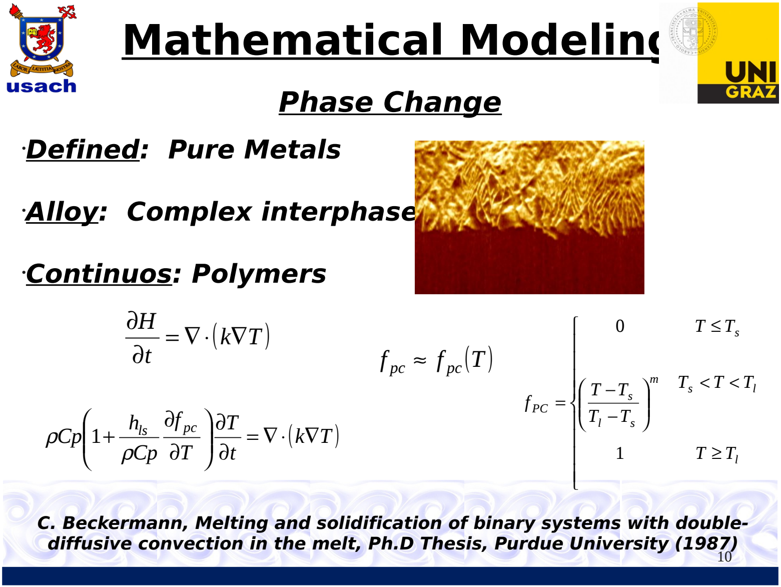


0	0	0	0	0	0	0.2	0.7
0	0	0	0.2	0.6	0.8	1	1
0	0	0	0.6	1	1	1	
0	0	0	0.2	0.7	1	1	
0	0	0	0	0.3			
0	0.1	0.2	0.3	0.7	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1

Position of the interphase

▨	**Full**	$F = 1$
▨	**Partially full**	$0 < F < 1$
□	**Empty**	$F = 0$

M. Kim, W. Lee International Journal for Numerical Methods in Fluids 42 (2003) 765-790.



Mathematical Modeling

Phase Change

- ***Defined*: Pure Metals**
- ***Alloy*: Complex interphase**
- ***Continuos*: Polymers**

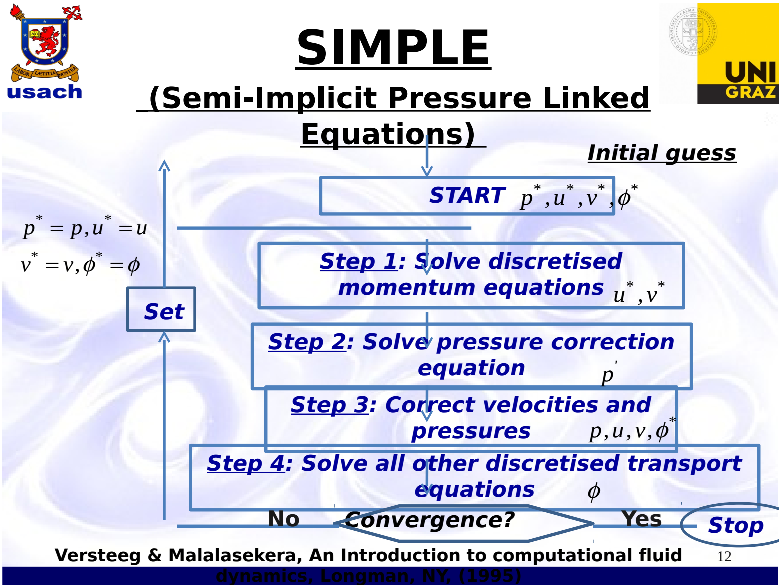
$$\frac{\partial H}{\partial t} = \nabla \cdot \left(k \nabla T\right)$$

$$f_{pc} \approx f_{pc}\left(T\right)$$

$$\rho Cp\left(1+\frac{h_{ls}}{\rho Cp}\frac{\partial f_{pc}}{\partial T}\right)\frac{\partial T}{\partial t} = \nabla \cdot \left(k \nabla T\right)$$

$$f_{PC} = \begin{cases} 0 & T \le T_s \\ \left(\dfrac{T - T_s}{T_l - T_s}\right)^m & T_s < T < T_l \\ 1 & T \ge T_l \end{cases}$$

C. Beckermann, Melting and solidification of binary systems with double-diffusive convection in the melt, Ph.D Thesis, Purdue University (1987)



SIMPLE

(Semi-Implicit Pressure Linked Equations)

Initial guess

START p^*, u^*, v^*, ϕ^*

Step 1: Solve discretised momentum equations u^*, v^*

Step 2: Solve pressure correction equation p'

Step 3: Correct velocities and pressures p, u, v, ϕ^*

Step 4: Solve all other discretised transport equations ϕ

Convergence?

No

Set

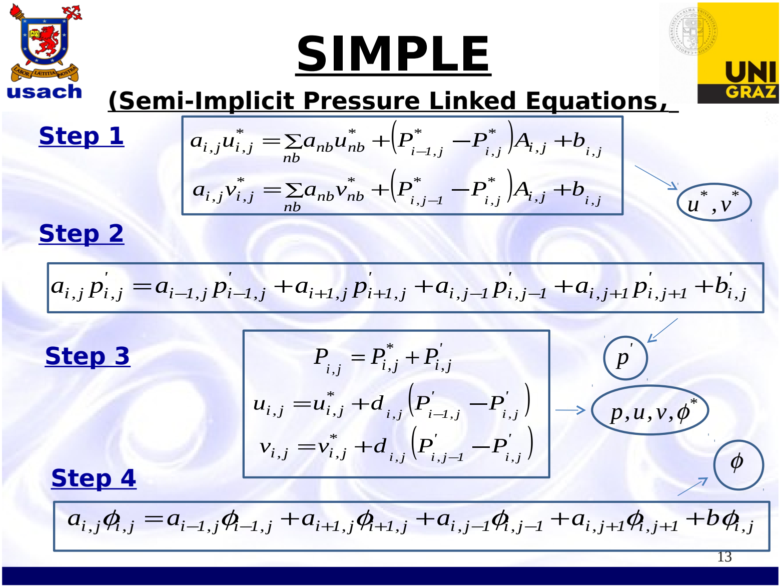
$p^* = p, u^* = u$

$v^* = v, \phi^* = \phi$

Yes

Stop

Versteeg & Malalasekera, An Introduction to computational fluid dynamics, Longman, NY, (1995)



SIMPLE

(Semi-Implicit Pressure Linked Equations,

Step 1

$$a_{i,j}u^*_{i,j} = \sum_{nb} a_{nb}u^*_{nb} + \left(P^*_{i-1,j} - P^*_{i,j}\right)A_{i,j} + b_{i,j}$$

$$a_{i,j}v^*_{i,j} = \sum_{nb} a_{nb}v^*_{nb} + \left(P^*_{i,j-1} - P^*_{i,j}\right)A_{i,j} + b_{i,j}$$

$$u^*, v^*$$

Step 2

$$a_{i,j}p'_{i,j} = a_{i-1,j}p'_{i-1,j} + a_{i+1,j}p'_{i+1,j} + a_{i,j-1}p'_{i,j-1} + a_{i,j+1}p'_{i,j+1} + b'_{i,j}$$

$$p'$$

Step 3

$$P_{i,j} = P^*_{i,j} + P'_{i,j}$$

$$u_{i,j} = u^*_{i,j} + d_{i,j}\left(P'_{i-1,j} - P'_{i,j}\right)$$

$$v_{i,j} = v^*_{i,j} + d_{i,j}\left(P'_{i,j-1} - P'_{i,j}\right)$$

$$p, u, v, \phi^*$$

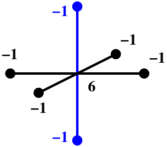
$$\phi$$

Step 4

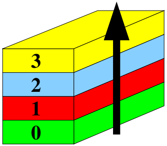
$$a_{i,j}\phi_{i,j} = a_{i-1,j}\phi_{i-1,j} + a_{i+1,j}\phi_{i+1,j} + a_{i,j-1}\phi_{i,j-1} + a_{i,j+1}\phi_{i,j+1} + b\phi_{i,j}$$

Solve linear system in each inner step

- system of coupled non-linear second order PDEs $\overset{\text{SIMPLE}}{\Longrightarrow}$ sequence of linear PDEs.
- unit cube, 7-point difference stencil



- Gauss-Seidel (forw/backw) wrt. plains in z-direction and
- ADI (Alternating Directions Iterative methods) in each plain
- shm parallel: combine plaines to a block Jacobi with above Gauss-Seidel in each block.



shm: system solve - naive approach

Listing 6: block-Jacobi Gauss-Seidel (shuffling)

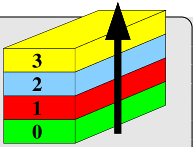
```
...
  !$omp parallel do shared(app) schedule(static)
    do k = 1, N                       //  plain k: forward
       .....
       app(i,j,k)  = ....             //  cube data
    end do
                                      //

 !$omp parallel do shared(app) schedule(static)
    do k = N, 1, -1                   //  plain k: backward
       .....
       app(i,j,k) += ....             //  cube data
    end do
...
```

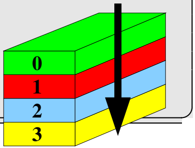
slower than on one thread

shm: system solve - naive approach

Listing 7: block-Jacobi Gauss-Seidel (shuffling)

```
...
  !$omp parallel do shared(app) schedule(static)
    do k = 1, N                        //  plain k
       .....
       app(i,j,k)  = ....              //  cube data
    end do
                                       //  data shuffling !!!!!!!

 !$omp parallel do shared(app) schedule(static)
    do k = N, 1, -1                    //  plain k
       .....
       app(i,j,k) += ....              //  cube data
    end do
...
```





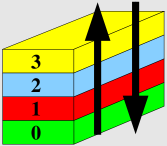
slower than on one thread

⟸ data blocks are remapped onto threads (data transfer!!)

shm: system solve - better approach

Listing 8: block-Jacobi Gauss-Seidel (no shuffling)

```
...
  !$omp parallel shared(app) schedule(static)
  nthrd = omp_get_num_threads()       // number of threads
  tid   = omp_get_thread_num()        // my thread ID
  lsize = int((kend-kst+1)/nthrds)+1  // junk size

  kf = tid*lsize + kst            // index range for this thread
  kl = min(((tid+1)*lsize + kst - 1),kend)
  kp = 1                          // first forward direction
  do nswz = 1,2
    do k = kf, kl, kp             //  plain k
       .....
       app(i,j,k)  = ....         //  cube data
    end do

    kp = -kp                      // reverse direction
  end do
```



shm: system solve - speedup

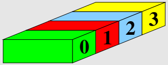
#threads	PROGRAM	SOLVE	MECFLU	SIMPLE	CALPH
1	298	85	186	30	51
2	188	54	116	20	32
4	101	28	63	12	17
6	73	18	45	9.2	10
8	61	15	37	8.6	66
12	47	10	27	8.6	7.5
speedup	**6.3**	**8.4**	6.8	**3.6**	6.7

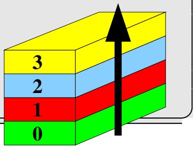
Speedup on dualhex, time in min.

- good speedup of 8.4 in SOLVE
- poor speedup of 3.6 in SIMPLE
  $*$ unnecessary reduce directive for an array (OpenMP 3.0)
  $*$ extra parallel–loop for boundary data $\Longrightarrow$ data shuffling
- speedup of 6.7 in update (vectors and material coeff.) can be further improved by avoiding above data shuffling

Listing 9: Handling of boundary data (shuffling)

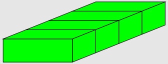
```
...
 !$omp parallel do shared(app) schedule(static)
    do j=1,N                           // lines  in plane
       ....
       app(i,j,1) += ....              // boundary data in plain 1
    end do


                                       //  data shuffling !!!!!!!
 !$omp parallel do shared(app) schedule(static)
    do k=1,N                           //  plain k
       .....
       app(i,j,k) += ....              //  cube data
    end do
...
```

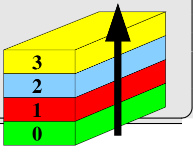




Listing 10: Handling of boundary data (no shuffling)

```
...
 !$omp parallel shared(app) schedule(static)
 tid = omp_get_thread_num()
 if (tid .EQ. 0)              // app(*,*,1)  stored on thread 0
    do j=1,N
       ....
       app(i,j,1) += ....     // boundary data
    end do
 end if
                              //  no data shuffling
 !$omp do
    do k=1,N                  //  plain k
       .....
       app(i,j,k) += ....     //  cube data
    end do
...
```





shm: system solve - Aug. 2011

#threads	PROGRAM	SOLVE	MECFLU	SIMPLE	CALPH
speedup July'11	**6.3**	**8.4**	6.8	**3.6**	6.7
speedup Aug'11	**11.7**	**11.4**	11.8	**11.3**	11.1

dualhex: Speedup with 12 threads

- no reduce arrays (SIMPLE)
- sequential handling of boundary data (no data shuffling)
- **temp. data** are always **private** (MECFLU)!
- no dynamic memory allocation in threads
- larger scope for #pragma omp parallel (PRAGMA)
- beware of **data race** in **loop dependencies** (↓) for pre-computed data

Nov. 2013: 4-year project in Chile for D. Vasco

shm: pitfall for polynom: $p = \sum_{k=0}^{N} a_k \cdot x^k$

Listing 11: Polynom sequentially

```
  p  = 0.0
  xk = 1.0
  do k = 1, N+1
     p  = p + a(k)*xk        ! add   a_k * x^k
     xk = xk*x
  end do
```

Listing 12: Polynom shm (wrong result)

```
  p  = 0.0
  xk = 1.0
!$omp parallel do private(k) shared(xk,a) reduction(+:p)
  do k = 1, N+1
     p  = p + a(k)*xk
     xk = xk*x               ! dependency between loops
  end do
```

shm: correct for polynom

$$p = \sum_{k=0}^{N} a_k \cdot x^k = \sum_{\text{tid}=0}^{\text{nthrds-1}} \sum_{k=kf_{\text{tid}}}^{kl_{\text{tid}}} a_k \cdot x^k$$

Calculate for thread `tid` its index range `[kf,kl]` explicitely.

Listing 13: Polynom shm

```
  p  = 0.0
!$omp parallel private(k,xk) shared(a) reduction(+:p)
  nthrd = omp_get_num_threads() ! number of threads
  tid   = omp_get_thread_num()  ! my thread number
  lsize = int((N+1)/nthrds)+1   ! my max. portion of data
  kf    = tid*lsize+1           ! interval
  kl    = min(kf+lsize, N+1)
  xk  = x**(kf-1)               ! correct x^k for this thread
  do k = kf, kl
     p  = p + a(k)*xk
     xk = xk*x
  end do
!$omp parallel                  ! correct result
```

Examples: programming environment

Getting the code

- Download code (link)
- unzip: `> tar xzf Chile.tgz`
- change into an environment *shm*: `> cd shm`
- change into a subdirectory: `> cd skalar`
- compile, link and run: `> make run`

Each directory contains at least *skalar* and jacobi, partially also with its MPI paralleliazion therein.

- `> cd shm; ls *default.mk` lists all supported compilers (here: GCC_ , ICC_ , PGI_)
- `> cd skalar; make COMPILER=ICC_ run` uses the Intel-compiler.

Supported parallel environments

- Sequential in directory *seq*.
- OpenMP 3.0 in directory *shm*.
- MPI in directory *par*.
- CUDA in directory *CUDA*.
- OpenACC in directory *OpenACC*.
- MIC (MIC-pragmas / OpenMP 4.0) in directory *MIC*.
- MPI+OpenMP in directory *OpenACC/par**.
- MPI+CUDA in *CUDA/par**.
- MPI+OpenACC in *OpenACC/par**: