Vectorisation and parallelisation of the neutron transport sweep algorithm on cartesian and hexagonal meshes using Kokkos

Café CExA, 3 Juin 2024

G. Suau [1] (gabriel.suau@cea.fr) R. Baron [2] A. Calloo [2] T. Gautier [3] R. Le Tellier [4]

[1] CEA DES/ISAS/DM2S/SERMA/LLPR

[2] CEA DES/ISAS/DM2S/SGLS/LCAN

[3] Inria / AVALON / LIP - ENS de Lyon

[4] CEA DES/IRESNE/DTN/STMA/LMAG

Numerical context

Sweep algorithm and parallelisation opportunities

Explicit vectorisation using Kokkos SIMD types

- Method and implementation
- Performance results

Parallelisation for multicore CPUs

- Sychronous parallel sweep using `RangePolicy`
- Adding angleset parallelism
- Adding asynchronicity with `Tasks` and `WorkGraphPolicy`
- Adding local work queues and work-stealing in `WorkGraphPolicy`

Conclusion and perspectives

Numerical context

Stationnary neutron transport equation

$$\left(\vec{\Omega}\cdot\vec{\nabla}+\Sigma_t(\vec{r},E)\right)\psi(\vec{r},E,\vec{\Omega}) = \int_{\mathcal{E}}\int_{\mathbb{S}^2}\Sigma_s(\vec{r},E'\leftarrow E,\vec{\Omega}'\leftarrow\vec{\Omega})\psi(\vec{r},E',\vec{\Omega}')d\vec{\Omega}'dE' + \frac{\chi(E)}{4\pi k_{eff}}\int_{\mathcal{E}}\nu\Sigma_f(\vec{r},E')\phi(\vec{r},E')dE', \quad (1)$$

$$\phi(\vec{r},E) = \int_{\mathbb{S}^2}\psi(\vec{r},E,\vec{\Omega})d\vec{\Omega}.$$

Energy and angular discretisations

- Multigroup formulation : n_g energy groups, $\mathcal{E} \to E_{ng} < \cdots < E_1 < E_0$;
- Discrete ordinates method (S_N) : n_d discrete directions on $\mathbb{S}^2$, quadrature formula $(\omega_d, \vec{\Omega}_d)$;
- Nested iterative algorithms : power iteration, Gauss-Seidel/Jacobi, Richardson;
- For each innermost iteration, solve

$$\forall g \in [\![1, n_g]\!] \begin{cases} \left(\vec{\Omega}_d\cdot\vec{\nabla}+\Sigma_t^g(\vec{r})\right)\psi_d^g(\vec{r}) = q_d^g(\vec{r}) \quad \forall d \in [\![1, n_d]\!] \\ \phi^g(\vec{r}) = \sum_{d=1}^{n_d}\omega_d\psi_d^g(\vec{r}). \end{cases} \quad (2)$$

Numerical context

Spatial discretisation

- For each innermost iteration, solve $\forall(g, d)$

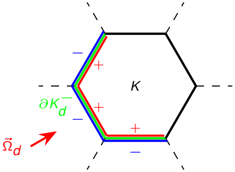
$$\vec{\Omega}_d \cdot \vec{\nabla}\psi_d^g(\vec{r}) + \Sigma_t^g(\vec{r})\psi_d^g(\vec{r}) = q_d^g(\vec{r}) \quad \forall \vec{r} \in \mathcal{D}. \tag{3}$$

- *Upwind Discontinuous Galerkin* $\longrightarrow$ meshing of the spatial domain, polynomial basis of order p in each cell K

$$\psi_{d|K}^g(\vec{r}) = \sum_{i=1}^{n(p)} \psi_{d,i}^g v_K^i(\vec{r}) = \underline{\psi}_{d,K}^g \cdot \underline{v}_K \tag{4}$$

- Discrete local formulation on a single (K, d, g)

$$\Big(\Omega_d^x \mathbf{A}_K^x + \Omega_d^y \mathbf{A}_K^y + \Omega_d^z \mathbf{A}_K^z + \Sigma_t^g \mathbf{M}_K - \sum_{F \in \partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F)\mathbf{M}_{K,F}^+\Big)\underline{\psi}_{d,K}^g$$
$$= \mathbf{M}_K \underline{q}_{K,d}^g - \sum_{F \in \partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F)\mathbf{M}_{K,F}^- \underline{\psi}_{d,K^F,-}^g \tag{5}$$



Elem. matrices

$$\left(\mathbf{M}_K\right)_{i,j} = \int_K v_K^i v_K^j dV,$$

$$\left(\mathbf{A}_K^*\right)_{i,j} = \int_K v_K^i \frac{\partial v_K^j}{\partial *} dV,$$

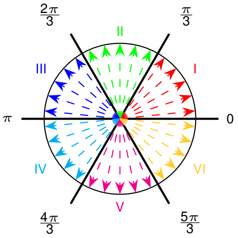
$$\left(\mathbf{M}_{K,F}^+\right)_{i,j} = \int_F v_K^i v_K^j dS,$$

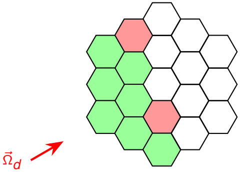
$$\left(\mathbf{M}_{K,F}^-\right)_{i,j} = \int_F v_K^i v_{K^F,-}^j dS.$$

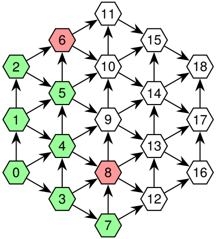
Sweep algorithm and parallelisation opportunities

Resolution via an ordered sweep of the mesh cells

- For each direction ($\mathcal{O}(10^{1-2})$) and energy group ($\mathcal{O}(10^{1-3})$), sweep mesh ($\mathcal{O}(10^{3-5})$),
- Can be seen as the traversal of a Directed Acyclic Graph (DAG), where each node is a cell,
- For each (K, d, g), assembly and resolution of a small linear system ($\mathcal{O}(10^{0-2})$) to compute spatial dofs,
- Directions and energy groups are independent $\rightarrow$ embarassingly parallel,
- Parallelism on cells is also available, although harder to tackle,
- Assembly and resolution of the local system can be parallelised.







Sweep algorithm and parallelisation opportunities

Algorithm: Pseudo-code for the angle-space-energy sweep

parallel for $s \in [\![1, n_s]\!]$ **do**
  parallel for $g \in [\![1, n_g]\!]$ **do**
    parallel for $d \in [\![1, n_d(s)]\!]$ **do**
      parallel graph $K \in \mathcal{G}$ **do**
        $\mathbf{C}^g_{d,K} = \vec{\Omega}_d \cdot \vec{\mathbf{A}}_K + \Sigma^g_{t,K}\mathbf{M}_K - \sum_{F\in\partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F)\mathbf{M}^+_{K,F}$ `// Local Matrix`
        $\underline{b}^g_{d,K} = \mathbf{M}_K \underline{q}^g_{d,K} - \sum_{F\in\partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F)\mathbf{M}^-_{K,F}\underline{\psi}^g_{d,K^{F,-}}$ `// Local RHS`
        $\underline{\psi}^g_{d,K} = (\mathbf{C}^g_{d,K})^{-1}\underline{b}^g_{d,K}$ `// Gaussian elimination`
      end
    end
  end
end

Sweep algorithm and parallelisation opportunities

Algorithm: Pseudo-code for the angle-space-energy sweep

parallel for $s \in [\![1, n_s]\!]$ **do**
  parallel for $d \in [\![1, n_d(s)]\!]$ **do**
    parallel for $g \in [\![1, n_g]\!]$ **do**
      parallel graph $K \in \mathcal{G}$ **do**
        $\mathbf{C}^g_{d,K} = \vec{\Omega}_d \cdot \vec{\mathbf{A}}_K + \Sigma^g_{t,K}\mathbf{M}_K - \sum_{F \in \partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F)\mathbf{M}^+_{K,F}$ `// Local Matrix`
        $\underline{b}^g_{d,K} = \mathbf{M}_K \underline{q}^g_{d,K} - \sum_{F \in \partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F)\mathbf{M}^-_{K,F} \underline{\psi}^g_{d,K^{F,-}}$ `// Local RHS`
        $\underline{\psi}^g_{d,K} = (\mathbf{C}^g_{d,K})^{-1} \underline{b}^g_{d,K}$ `// Gaussian elimination`
      end
    end
  end
end

Sweep algorithm and parallelisation opportunities

Algorithm: Pseudo-code for the angle-space-energy sweep

```
parallel for s ∈ ⟦1, n_s⟧ do
    parallel graph K ∈ 𝒢 do
        parallel for g ∈ ⟦1, n_g⟧ do
            parallel for d ∈ ⟦1, n_d(s)⟧ do
                C^g_{d,K} = Ω_d · A_K + Σ^g_{t,K} M_K − Σ_{F∈∂K_d^−} (Ω_d · n_F) M^+_{K,F}   // Local Matrix
                b^g_{d,K} = M_K q^g_{d,K} − Σ_{F∈∂K_d^−} (Ω_d · n_F) M^−_{K,F} ψ^g_{d,K^{F,−}}   // Local RHS
                ψ^g_{d,K} = (C^g_{d,K})^{−1} b^g_{d,K}   // Gaussian elimination
            end
        end
    end
end
```

$$\mathbf{C}^g_{d,K} = \vec{\Omega}_d \cdot \vec{\mathbf{A}}_K + \Sigma^g_{t,K}\mathbf{M}_K - \sum_{F\in\partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F)\mathbf{M}^+_{K,F}$$

$$\underline{b}^g_{d,K} = \mathbf{M}_K \underline{q}^g_{d,K} - \sum_{F\in\partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F)\mathbf{M}^-_{K,F}\underline{\psi}^g_{d,K^{F,-}}$$

$$\underline{\psi}^g_{d,K} = (\mathbf{C}^g_{d,K})^{-1}\underline{b}^g_{d,K}$$

Sweep algorithm and parallelisation opportunities

Algorithm: Pseudo-code for the angle-space-energy sweep

parallel for $s \in [\![1, n_s]\!]$ **do**
  parallel graph $K \in \mathcal{G}$ **do**
    parallel for $d \in [\![1, n_d(s)]\!]$ **do**
      parallel for $g \in [\![1, n_g]\!]$ **do**
        $\mathbf{C}^g_{d,K} = \vec{\Omega}_d \cdot \vec{\mathbf{A}}_K + \Sigma^g_{t,K}\mathbf{M}_K - \sum_{F \in \partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F)\mathbf{M}^+_{K,F}$ `// Local Matrix`
        $\underline{b}^g_{d,K} = \mathbf{M}_K \underline{q}^g_{d,K} - \sum_{F \in \partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F)\mathbf{M}^-_{K,F}\underline{\psi}^g_{d,K^{F,-}}$ `// Local RHS`
        $\underline{\psi}^g_{d,K} = (\mathbf{C}^g_{d,K})^{-1}\underline{b}^g_{d,K}$ `// Gaussian elimination`
      end
    end
  end
end

Sweep algorithm and parallelisation opportunities

Algorithm: Pseudo-code for the angle-space-energy sweep

parallel for $s \in [\![1, n_s]\!]$ **do**
  parallel graph $K \in \mathcal{G}$ **do**
    parallel for $d \in [\![1, n_d(s)]\!]$ **do**
      $\mathbf{T}_{d,K} = \vec{\Omega}_d \cdot \vec{\mathbf{A}}_K - \sum_{F \in \partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F) \mathbf{M}^+_{K,F}$ `// Start mat. assembly`
      parallel for $g \in [\![1, n_g]\!]$ **do**
        $\mathbf{C}^g_{d,K} = \mathbf{T}_{d,K} + \Sigma^g_{t,K} \mathbf{M}_K$ `// Finish mat. assembly`
        $\underline{b}^g_{d,K} = \mathbf{M}_K \underline{q}^g_{d,K} - \sum_{F \in \partial K_d^-} (\vec{\Omega}_d \cdot \vec{n}_F) \mathbf{M}^-_{K,F} \underline{\psi}^g_{d,K^{F,-}}$ `// Local RHS`
        $\underline{\psi}^g_{d,K} = (\mathbf{C}^g_{d,K})^{-1} \underline{b}^g_{d,K}$ `// Gaussian elimination`
      end
    end
  end
end

Sweep algorithm and parallelisation opportunities

Simplified C++/Kokkos sequential implementation

```
LayoutRight lay(nas, nc, nd, ng, nm);
View<float*****, LayoutRight> psi("psi", lay), src("src", lay);
View<float**   , LayoutRight> T('T', nm, nm), C('C', nm, nm);
View<float*    , LayoutRight> b('b', nm);
for (int as = 0; as < nas; ++as) {
  for (int K = cells.begin(); K != cells.end(); cells.next()) {
    for (int d = 0; d < nd; ++d) {
      build_Tmat(T, ...);      // Begin matrix assembly
      for (int g = 0; g < ng; ++g) {
        auto loc_psi = subview(psi, as, K, d, g, ALL);
        build_Cmat(C, T, ...); // Finish matrix assembly
        build_brhs(b, ...);    // Do RHS assembly
        solve(C, loc_psi, b);  // Solve
      }
    }
  }
}
```

Explicit vectorisation using Kokkos SIMD types

Method and implementation

Why ?

- Maximise single-core performance on CPU

Where ?

- Linear system assembly/solve
  $\Longrightarrow$ auto-vectorisation, compiler-dependent
  $\Longrightarrow$ might not be efficient for small system sizes
- Loop on groups
  $\Longrightarrow$ n_g is HIGHLY problem-dependent
  $\Longrightarrow$ small n_g leads to poor vectorisation performance
- Loop on directions
  $\Longrightarrow$ a few dozens directions per angleset
  $\Longrightarrow$ completely independent
  $\Longrightarrow$ chosen strategy

How ?

- Auto-vectorisation $\Longrightarrow$ requires adjusting memory layout to be contiguous in directions + compiler-dependent,
- Intrinsics $\Longrightarrow$ complex and not portable code,
- SIMD types $\Longrightarrow$ easy to read AND portable code !

Explicit vectorisation using Kokkos SIMD types

Method and implementation

Maximum speedup

- n_d = num. directions per angleset, n_{pad} = num. dummy directions, V = SIMD size,

$$S_{max} = V \times n_d/(n_d + n_{pad}) \tag{6}$$

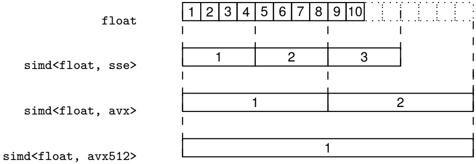


Figure 2: Illustration of the padding strategy with 10 directions per angleset for SSE, AVX and AVX512 in single precision.

Explicit vectorisation using Kokkos SIMD types

Method and implementation

Simplified C++/Kokkos SIMD implementation

```
using simd_t = simd<float, simd_abi::native>;
int ndv = (nd + simd_t::size() - 1) / simd_t::size();
LayoutRight lay(nas, nc, ndv, ng, nm);
View<simd_t*****, LayoutRight> psi("psi", lay), src("src", lay);
View<simd_t**   , LayoutRight> T('T', nm, nm), C('C', nm, nm);
View<simd_t*    , LayoutRight> b('b', nm);
for (int as = 0; as < nas; ++as) {
  for (int K = cells.begin(); K != cells.end(); cells.next()) {
    for (int d = 0; d < ndv; ++d) {
      build_Tmat(T, ...);       // Begin matrix assembly
      for (int g = 0; g < ng; ++g) {
        auto loc_psi = subview(psi, as, K, d, g, ALL);
        build_Cmat(C, T, ...); // Finish matrix assembly
        build_brhs(b, ...);    // Do RHS assembly
        solve(C, loc_psi, b);  // Solve
      }
    }
  }
}
```

Explicit vectorisation using Kokkos SIMD types

Performance results

Machine used for all performance tests

- One node composed of two 24-cores AVX-enabled AMD EPYC 7352 processors,
- 256 GB of memory, 8 NUMA node in total

Software configuration

- GCC 11.2.0, Kokkos 4.3.1,
- Enable OpenMP backend,
- Compiler options `-O2 -march=native, -mtune=native`

Description of the experiments

- (8, 8, 8) 3D cartesian mesh, 12 directions per angleset, 4 energy groups,
- Varying number of spatial dofs in $\{1, \ldots, 32\}$,
- Test with `float` and `simd<float, simd_abi::native>`.

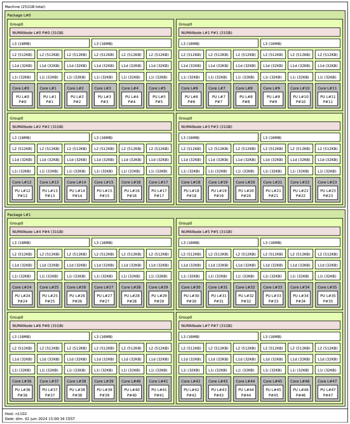


Figure 3: `lstopo` output

Explicit vectorisation using Kokkos SIMD types

Performance results

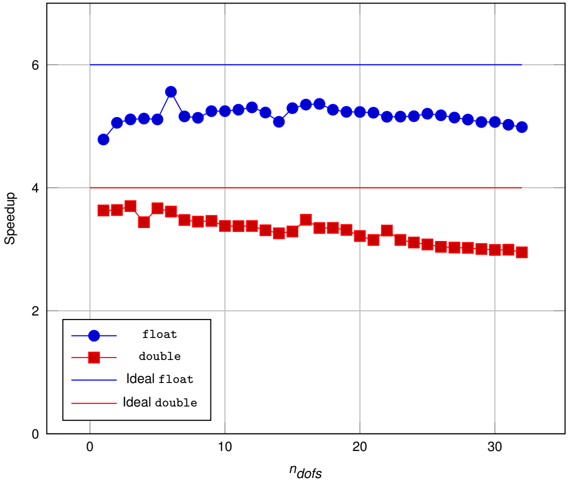


Figure 4: Speedup of the Kokkos SIMD implementation versus a scalar implementation. Ideal speedup may not be equal to the SIMD size due to the padding strategy.

Parallelisation for multicore CPUs

Sychronous parallel sweep using `RangePolicy`

Where can we parallelise ?

- Anglesets or directions $\Longrightarrow$ limited, too few directions
- Groups $\Longrightarrow$ unreliable, too problem dependent
- Cells $\Longrightarrow$ enough parallelism, must take care of the upwind dependencies
- Front-synchronous strategy : sequential loop on fronts, parallel loop inside each front

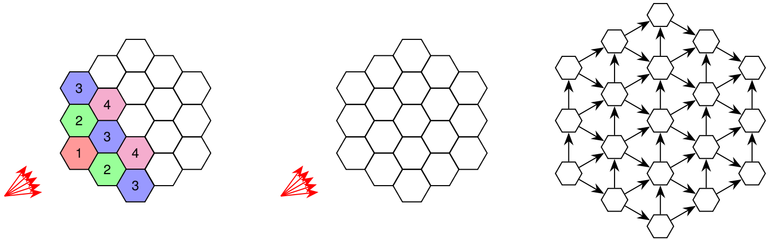


Figure 5: Front-synchronous sweep on a 2-rings hexagonal 2D mesh with 2 threads.

Parallelisation for multicore CPUs

Sychronous parallel sweep using `RangePolicy`

Where can we parallelise ?

- Anglesets or directions $\Longrightarrow$ limited, too few directions
- Groups $\Longrightarrow$ unreliable, too problem dependent
- Cells $\Longrightarrow$ enough parallelism, must take care of the upwind dependencies
- Front-synchronous strategy : sequential loop on fronts, parallel loop inside each front

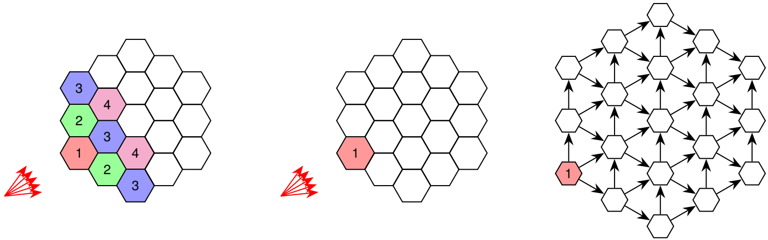


Figure 5: Front-synchronous sweep on a 2-rings hexagonal 2D mesh with 2 threads.

Parallelisation for multicore CPUs

Sychronous parallel sweep using `RangePolicy`

Where can we parallelise ?

- Anglesets or directions $\Longrightarrow$ limited, too few directions
- Groups $\Longrightarrow$ unreliable, too problem dependent
- Cells $\Longrightarrow$ enough parallelism, must take care of the upwind dependencies
- Front-synchronous strategy : sequential loop on fronts, parallel loop inside each front

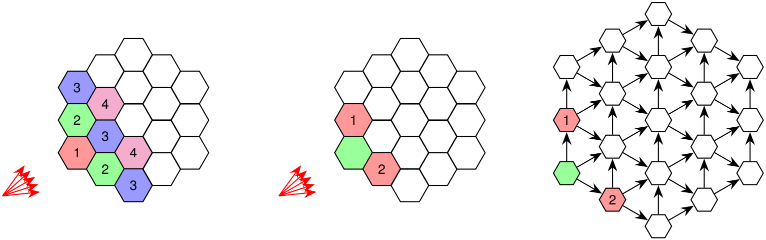


Figure 5: Front-synchronous sweep on a 2-rings hexagonal 2D mesh with 2 threads.

Parallelisation for multicore CPUs

Sychronous parallel sweep using `RangePolicy`

Where can we parallelise ?

- Anglesets or directions $\Longrightarrow$ limited, too few directions
- Groups $\Longrightarrow$ unreliable, too problem dependent
- Cells $\Longrightarrow$ enough parallelism, must take care of the upwind dependencies
- Front-synchronous strategy : sequential loop on fronts, parallel loop inside each front

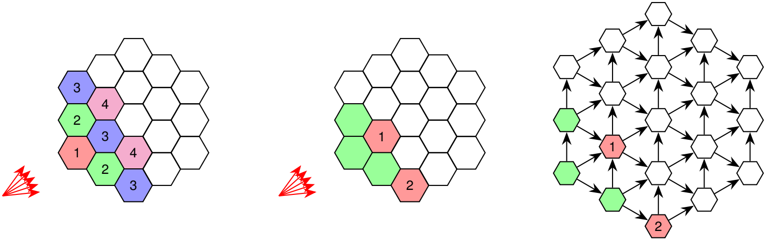


Figure 5: Front-synchronous sweep on a 2-rings hexagonal 2D mesh with 2 threads.

Parallelisation for multicore CPUs

Sychronous parallel sweep using `RangePolicy`

Where can we parallelise ?

- Anglesets or directions $\Longrightarrow$ limited, too few directions
- Groups $\Longrightarrow$ unreliable, too problem dependent
- Cells $\Longrightarrow$ enough parallelism, must take care of the upwind dependencies
- Front-synchronous strategy : sequential loop on fronts, parallel loop inside each front

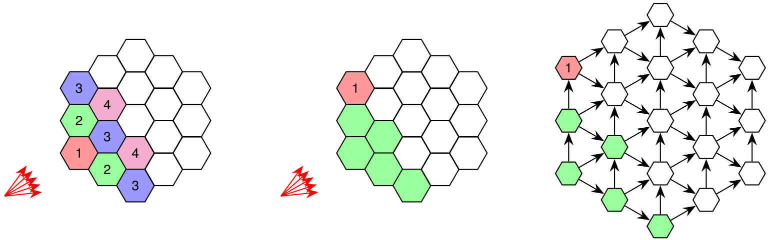


Figure 5: Front-synchronous sweep on a 2-rings hexagonal 2D mesh with 2 threads.

Parallelisation for multicore CPUs

Sychronous parallel sweep using `RangePolicy`

Where can we parallelise ?

- Anglesets or directions $\Longrightarrow$ limited, too few directions
- Groups $\Longrightarrow$ unreliable, too problem dependent
- Cells $\Longrightarrow$ enough parallelism, must take care of the upwind dependencies
- Front-synchronous strategy : sequential loop on fronts, parallel loop inside each front

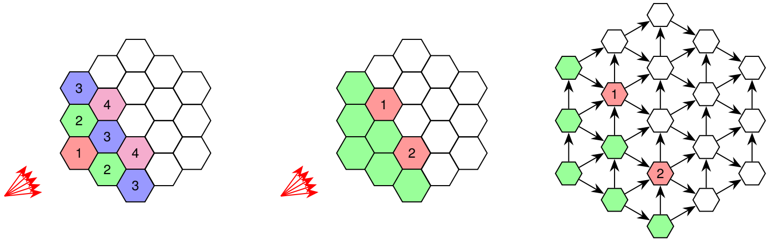


Figure 5: Front-synchronous sweep on a 2-rings hexagonal 2D mesh with 2 threads.

Parallelisation for multicore CPUs

Sychronous parallel sweep using `RangePolicy`

Front-synchronous implementation

```
for (int as = 0; as < nas; ++as) {
  for (int f = 0; f < nf; ++f) {
    auto ready = get_ready_cells(as, f);
    parallel_for(ready.size(), KOKKOS_LAMBDA (int fK) {
      int const K = ready(fK);
      for (int d = 0; d < ndv; ++d) {
        build_Tmat(T, ...);       // Begin matrix assembly
        for (int g = 0; g < ng; ++g) {
          auto loc_psi = subview(psi, as, K, d, g, ALL);
          build_Cmat(C, T, ...); // Finish matrix assembly
          build_brhs(b, ...);    // Do RHS assembly
          solve(C, loc_psi, b);  // Solve
        }
      }
    });
  }
}
```

Parallelisation for multicore CPUs

Sychronous parallel sweep using `RangePolicy`

Hexagonal 3D test case

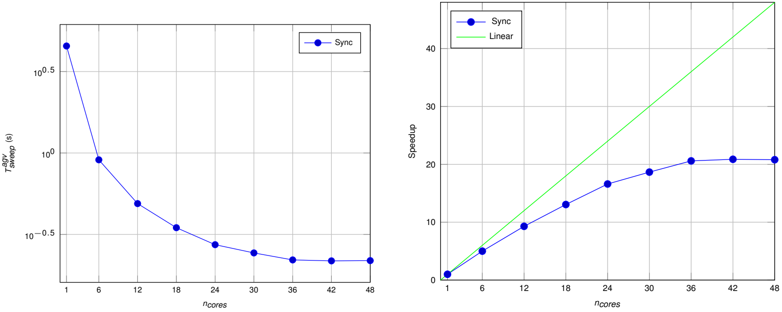
- TAKEDA-4 hexagonal benchmark
- 7 rings, 38 axial planes $\implies$ $169 \times 38 = 6422$ cells
- 12 anglesets, 20 directions per angleset, 4 energy groups,
- 20 spatial dofs per cell

Cartesian 3D test case

- $(32, 32, 32)$ cartesian grid $\implies$ 32768 cells
- 8 anglesets, 20 directions per angleset, 7 energy groups,
- 21 spatial dofs per cell

Parallelisation for multicore CPUs

Sychronous parallel sweep using `RangePolicy`



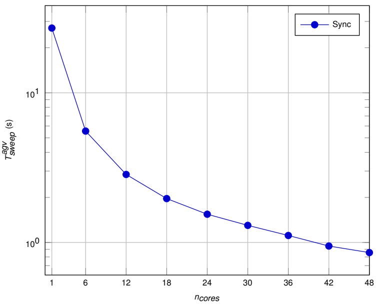
(a) Average sweep time (s)

(b) Speedup

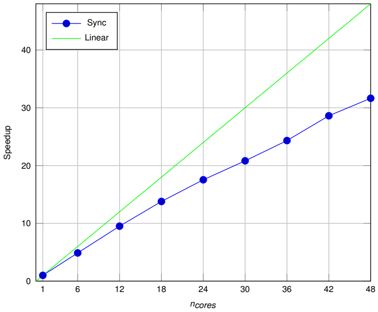
Figure 6: Hexagonal 3D test performance results

Parallelisation for multicore CPUs

Sychronous parallel sweep using `RangePolicy`



(a) Average sweep time (s)



(b) Speedup

Figure 7: Cartesian 3D test performance results

Parallelisation for multicore CPUs

Adding angleset parallelism

Why ?

- Expose more parallelism at each step
- Reduce thread idle time

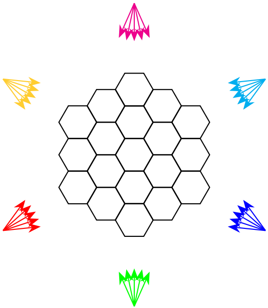


Figure 8: Front-synchronous sweep with added angleset parallelism on a 2-rings hexagonal 2D mesh with 4 threads.

Parallelisation for multicore CPUs

Adding angleset parallelism

Why ?

- Expose more parallelism at each step
- Reduce thread idle time

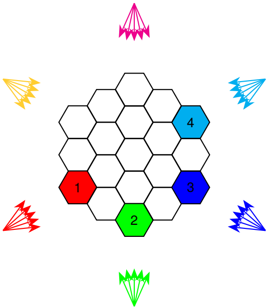


Figure 8: Front-synchronous sweep with added angleset parallelism on a 2-rings hexagonal 2D mesh with 4 threads.

Parallelisation for multicore CPUs

Adding angleset parallelism

Why ?

- Expose more parallelism at each step
- Reduce thread idle time

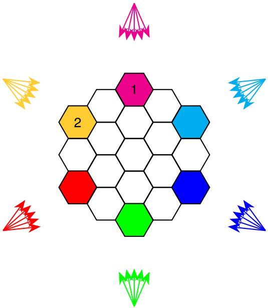


Figure 8: Front-synchronous sweep with added angleset parallelism on a 2-rings hexagonal 2D mesh with 4 threads.

Parallelisation for multicore CPUs

Adding angleset parallelism

Why ?

- Expose more parallelism at each step
- Reduce thread idle time

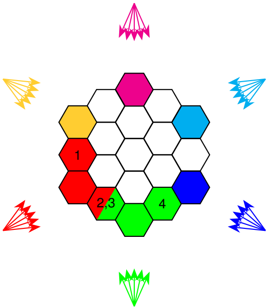


Figure 8: Front-synchronous sweep with added angleset parallelism on a 2-rings hexagonal 2D mesh with 4 threads.

Parallelisation for multicore CPUs

Adding angleset parallelism

Why ?

- Expose more parallelism at each step
- Reduce thread idle time

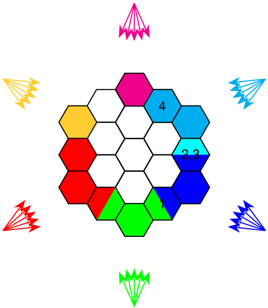


Figure 8: Front-synchronous sweep with added angleset parallelism on a 2-rings hexagonal 2D mesh with 4 threads.

Parallelisation for multicore CPUs

Adding angleset parallelism

Why ?

- Expose more parallelism at each step
- Reduce thread idle time

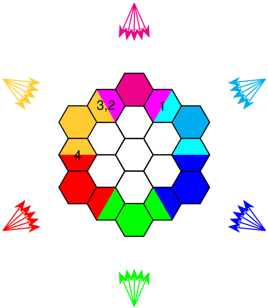


Figure 8: Front-synchronous sweep with added angleset parallelism on a 2-rings hexagonal 2D mesh with 4 threads.

Parallelisation for multicore CPUs

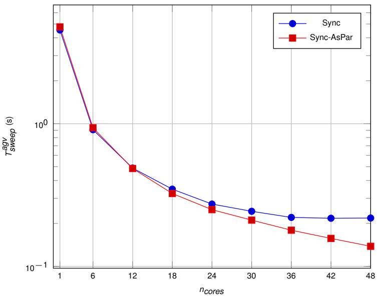
Adding angleset parallelism

Front-synchronous parallel anglesets implementation

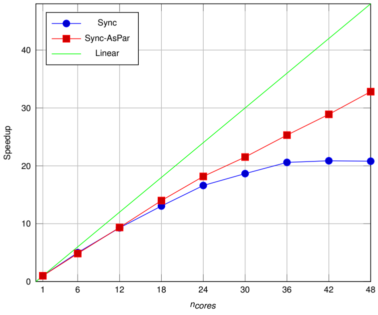
```
for (int f = 0; f < nf; ++f) {
  auto ready = get_ready_cells(f);
  parallel_for(ready.size(), KOKKOS_LAMBDA (int asfK) {
    auto const& [as, K] = ready(asfK);
    for (int d = 0; d < ndv; ++d) {
      build_Tmat(T, ...);       // Begin matrix assembly
      for (int g = 0; g < ng; ++g) {
        auto loc_psi = subview(psi, as, K, d, g, ALL);
        build_Cmat(C, T, ...); // Finish matrix assembly
        build_brhs(b, ...);    // Do RHS assembly
        solve(C, loc_psi, b);  // Solve
      }
    }
  });
}
```

Parallelisation for multicore CPUs

Adding angleset parallelism



(a) Average sweep time (s)

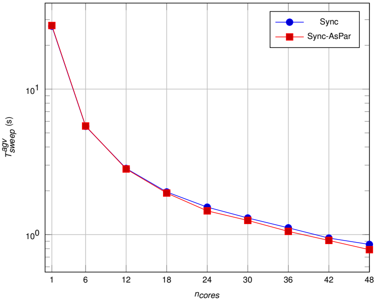


(b) Speedup

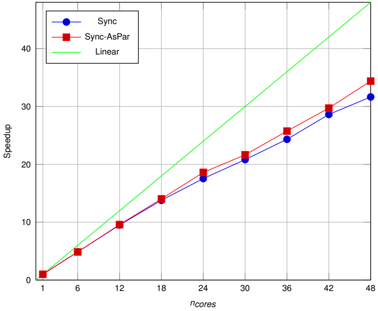
Figure 9: Hexagonal 3D test performance results

Parallelisation for multicore CPUs

Adding angleset parallelism



(a) Average sweep time (s)



(b) Speedup

Figure 10: Cartesian 3D test performance results

Parallelisation for multicore CPUs

Adding asynchronicity with `Tasks` and `WorkGraphPolicy`

Task parallelism

- One task = solve all directions and groups for a single cell for a given angleset
- $n_{tasks} = n_c \times n_{as}$

`WorkGraphPolicy`

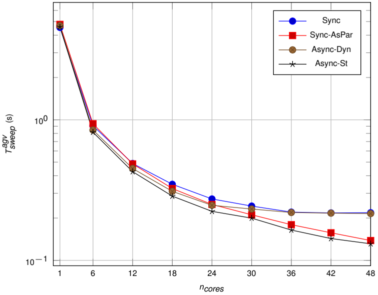
- Task-DAG = CSR graph, built once before the computation,
- Cost of launching task = atomic decrement of an integer $\Longrightarrow$ very lightweight

Dynamic `Tasks`

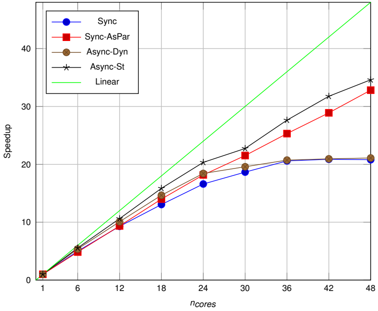
- Initially spawn n_{as} tasks (= first cell for each angleset)
- Tasks dynamically spawn other tasks
- Manually keep count of the dependency counts

Parallelisation for multicore CPUs

Adding asynchronicity with `Tasks` and `WorkGraphPolicy`



(a) Average sweep time (s)

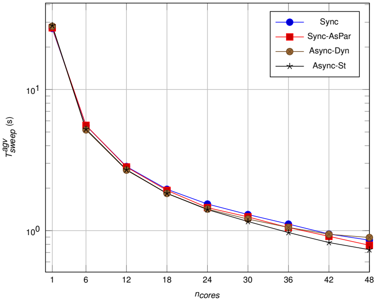


(b) Speedup

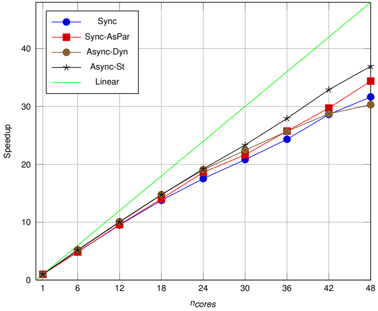
Figure 11: Hexagonal 3D test performance results

Parallelisation for multicore CPUs

Adding asynchronicity with `Tasks` and `WorkGraphPolicy`



(a) Average sweep time (s)



(b) Speedup

Figure 12: Cartesian 3D test performance results

Parallelisation for multicore CPUs

Adding local work queues and work-stealing in `WorkGraphPolicy`

Kokkos `WorkGraphPolicy` details

- Single work queue shared by all threads,
- FIFO structure, threads push work to the head, and pop from the tail,
- All accesses to the work queue are atomic,
- Single wating count queue shared by all threads, atomically updated at the end of each task

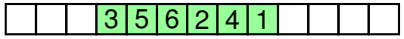


Figure 13: Global work-queue in `WorkGraphPolicy`

Parallelisation for multicore CPUs

Adding local work queues and work-stealing in `WorkGraphPolicy`

Kokkos `WorkGraphPolicy` details

- Single work queue shared by all threads,
- FIFO structure, threads push work to the head, and pop from the tail,
- All accesses to the work queue are atomic,
- Single wating count queue shared by all threads, atomically updated at the end of each task

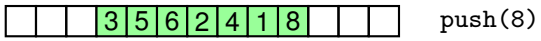


Figure 13: Global work-queue in `WorkGraphPolicy`

Parallelisation for multicore CPUs

Adding local work queues and work-stealing in `WorkGraphPolicy`

Kokkos `WorkGraphPolicy` details

- Single work queue shared by all threads,
- FIFO structure, threads push work to the head, and pop from the tail,
- All accesses to the work queue are atomic,
- Single wating count queue shared by all threads, atomically updated at the end of each task

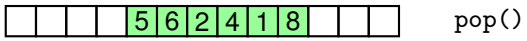


Figure 13: Global work-queue in `WorkGraphPolicy`

Parallelisation for multicore CPUs

Adding local work queues and work-stealing in `WorkGraphPolicy`

Custom `WorkGraphPolicy` details

- One work queue per thread,
- LIFO structure, a thread pushes to and pops from the head of its queue,
- No atomic operations needed for push and pop,
- When its queue is empty, a thread becomes a thief
  1. acquire a random victim's queue lock,
  2. try to steal victim's queue tail,
  3. on success, execute the task; on failure go to step 1;

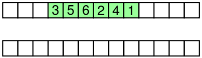


Figure 14: Local work-queues and work-stealing in `WorkGraphPolicyCustom`

Expected benefits

- Better locality : the last pushed task is more likely to reuse data from previous task (in cache)
- Less contention to push and pop tasks : only between a thief and its victim

Parallelisation for multicore CPUs

Adding local work queues and work-stealing in `WorkGraphPolicy`

Custom `WorkGraphPolicy` details

- One work queue per thread,
- LIFO structure, a thread pushes to and pops from the head of its queue,
- No atomic operations needed for push and pop,
- When its queue is empty, a thread becomes a thief
  1. acquire a random victim's queue lock,
  2. try to steal victim's queue tail,
  3. on success, execute the task; on failure go to step 1;

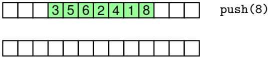


Figure 14: Local work-queues and work-stealing in `WorkGraphPolicyCustom`

Expected benefits

- Better locality : the last pushed task is more likely to reuse data from previous task (in cache)
- Less contention to push and pop tasks : only between a thief and its victim

Parallelisation for multicore CPUs

Adding local work queues and work-stealing in `WorkGraphPolicy`

Custom `WorkGraphPolicy` details

- One work queue per thread,
- LIFO structure, a thread pushes to and pops from the head of its queue,
- No atomic operations needed for push and pop,
- When its queue is empty, a thread becomes a thief
  1. acquire a random victim's queue lock,
  2. try to steal victim's queue tail,
  3. on success, execute the task; on failure go to step 1;

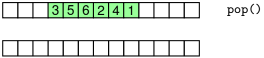


Figure 14: Local work-queues and work-stealing in `WorkGraphPolicyCustom`

Expected benefits

- Better locality : the last pushed task is more likely to reuse data from previous task (in cache)
- Less contention to push and pop tasks : only between a thief and its victim

Parallelisation for multicore CPUs

Adding local work queues and work-stealing in `WorkGraphPolicy`

Custom `WorkGraphPolicy` details

- One work queue per thread,
- LIFO structure, a thread pushes to and pops from the head of its queue,
- No atomic operations needed for push and pop,
- When its queue is empty, a thread becomes a thief
  1. acquire a random victim's queue lock,
  2. try to steal victim's queue tail,
  3. on success, execute the task; on failure go to step 1;

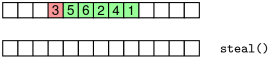


Figure 14: Local work-queues and work-stealing in `WorkGraphPolicyCustom`

Expected benefits

- Better locality : the last pushed task is more likely to reuse data from previous task (in cache)
- Less contention to push and pop tasks : only between a thief and its victim

Parallelisation for multicore CPUs

Adding local work queues and work-stealing in `WorkGraphPolicy`

Custom `WorkGraphPolicy` details

- One work queue per thread,
- LIFO structure, a thread pushes to and pops from the head of its queue,
- No atomic operations needed for push and pop,
- When its queue is empty, a thread becomes a thief
  1. acquire a random victim's queue lock,
  2. try to steal victim's queue tail,
  3. on success, execute the task; on failure go to step 1;

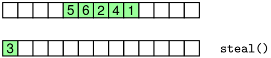


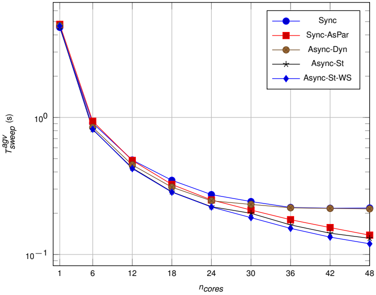
Figure 14: Local work-queues and work-stealing in `WorkGraphPolicyCustom`

Expected benefits

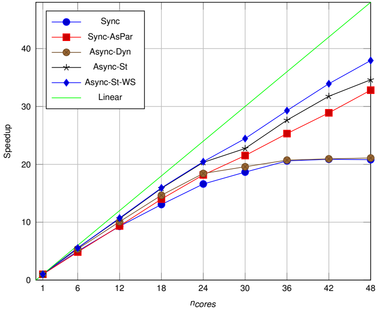
- Better locality : the last pushed task is more likely to reuse data from previous task (in cache)
- Less contention to push and pop tasks : only between a thief and its victim

Parallelisation for multicore CPUs

Adding local work queues and work-stealing in `WorkGraphPolicy`



(a) Average sweep time (s)

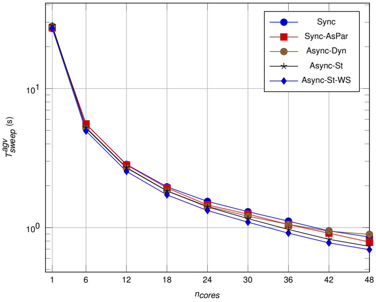


(b) Speedup

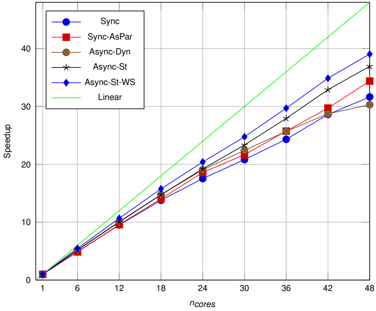
Figure 15: Hexagonal 3D test performance results

Parallelisation for multicore CPUs

Adding local work queues and work-stealing in `WorkGraphPolicy`



(a) Average sweep time (s)



(b) Speedup

Figure 16: Cartesian 3D test performance results

Conclusion and perspectives

What's been done

- Efficient and portable vectorised sweep implementation
- Task-based multicore implementation using modified Kokkos `WorkGraphPolicy` with work-stealing

Next steps

- Parametric study of the implementation (task size, number of tasks, number of threads)

What about GPU performance ?

- Very bad performance on GPUs,
- Preliminary results :
  - One thread per front cell gives very bad performance (uncoalesced accesses, not enough parallel work),
  - One thread per (K, d, g) front triplet is better, but still far from what we can expect (not enough parallel work, bad use of fast memory),
  - Need to parallelise the linear system assembly and resolution,
  - Use `TeamPolicy` with one team per linear system $\Longrightarrow$ 1 team per (K, d, g) triplet,
  - Need optimised batched linear albegra kernels.