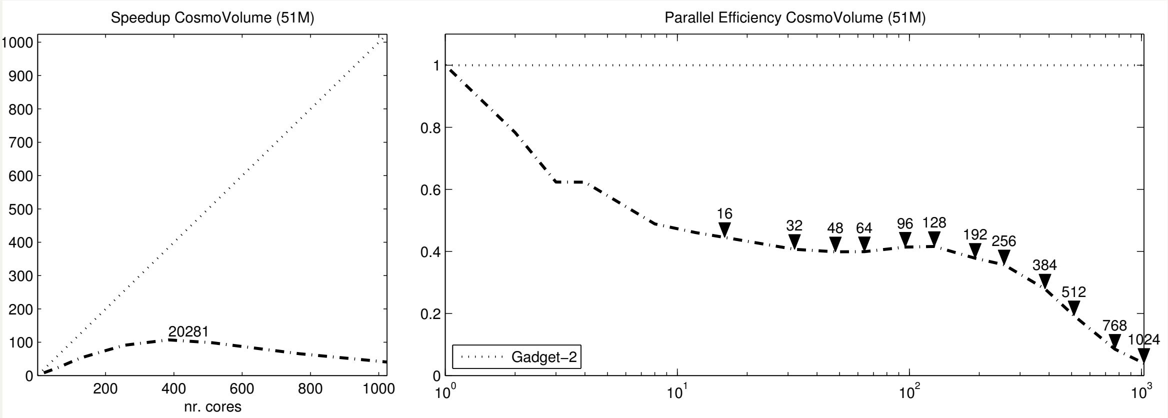
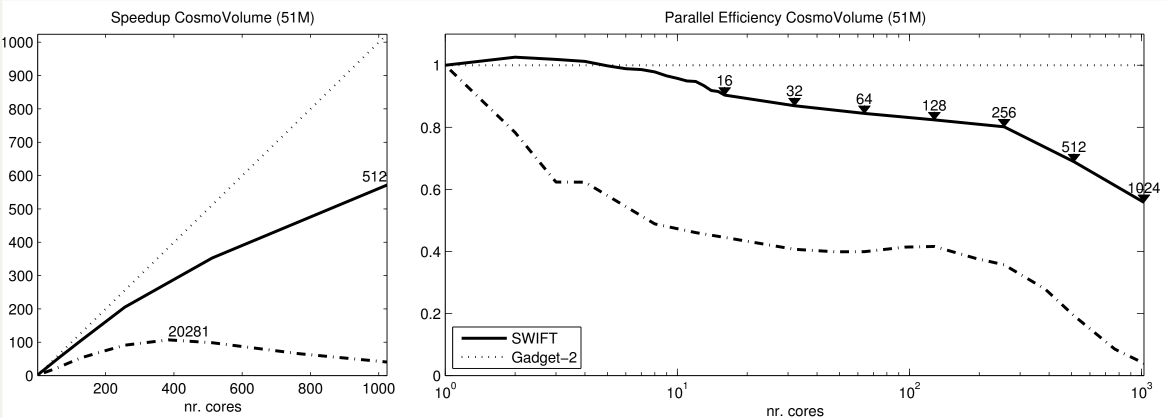
SWIFT: Task-based parallelism, hybrid shared/distributed-memory parallelism, and SPH simulations

Pedro Gonnet, Matthieu Schaller, Aidan Chalk, Tom Theuns
SECS/ICC, Durham University
Exascale Computing in Astrophysics, September 10th, 2013

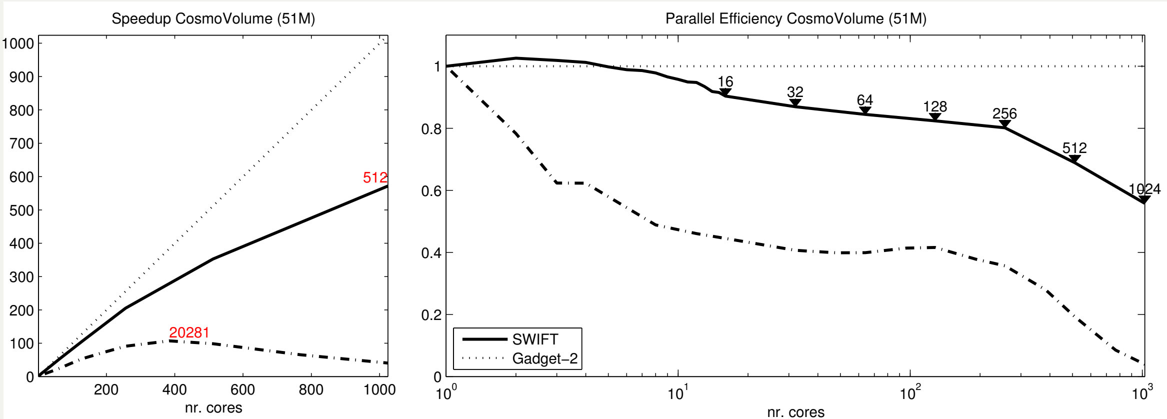
Introduction
This talk in a nutshell



- 51M particle EAGLE box ($z = 0.5$) SPH-only simulation on the COSMA5 cluster.



- 51M particle EAGLE box ($z = 0.5$) SPH-only simulation on the COSMA5 cluster.
- SWIFT: Strong scaling up to 1024 cores with 60% parallel efficiency.



- 51M particle EAGLE box ($z = 0.5$) SPH-only simulation on the COSMA5 cluster.
- SWIFT: Strong scaling up to 1024 cores with 60% parallel efficiency.
- $\sim 40\times$ faster than GADGET.

Introduction

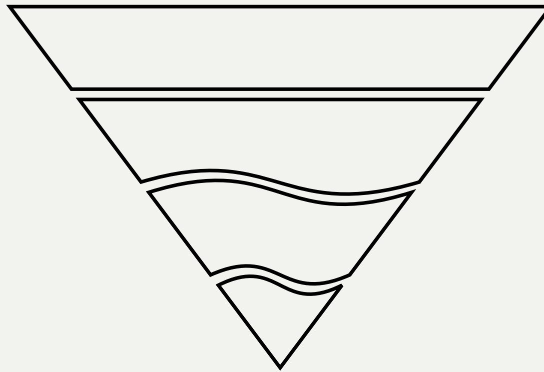
A hierarchy of contributions

Domain Science

Software

Algorithms

Paradigms



Introduction

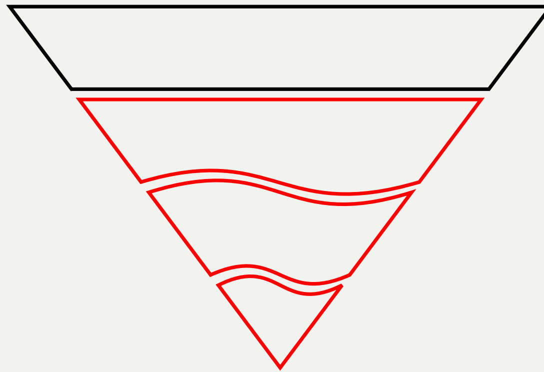
A hierarchy of contributions

Domain Science

Software

Algorithms

Paradigms



Task-based parallelism

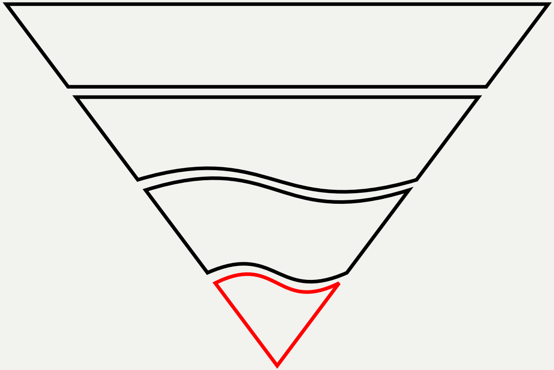
Replacing the paradigm

Domain Science

Software

Algorithms

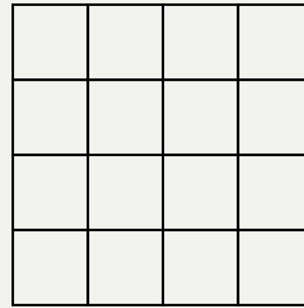
Paradigms



Task-based parallelism

The problem with distributed-memory parallelism

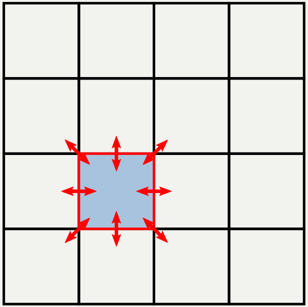
- Distributed-memory parallelism, e.g. using MPI, is based on data decomposition, i.e. each processor is assigned part of the problem to work on and communicates with its neighbours.
- Surface-to-volume ratio problem: As the number of cores increases, the amount of computation per core (volume) decreases while the relative amount of communication (surface) increases, eventually dominating the entire computation.
  $\longrightarrow$ We can always do larger simulations, but not smaller simulations faster.



Task-based parallelism

The problem with distributed-memory parallelism

- Distributed-memory parallelism, e.g. using MPI, is based on data decomposition, i.e. each processor is assigned part of the problem to work on and communicates with its neighbours.
- Surface-to-volume ratio problem: As the number of cores increases, the amount of computation per core (volume) decreases while the relative amount of communication (surface) increases, eventually dominating the entire computation.
  $\longrightarrow$ We can always do larger simulations, but not smaller simulations faster.

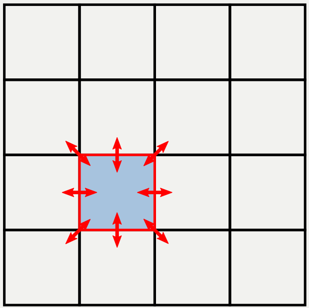


Task-based parallelism

The problem with distributed-memory parallelism

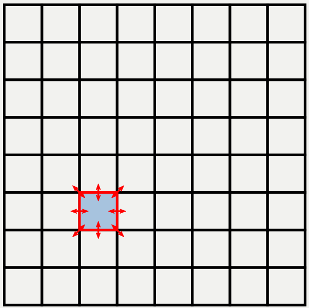
Durham University

- Distributed-memory parallelism, e.g. using MPI, is based on data decomposition, i.e. each processor is assigned part of the problem to work on and communicates with its neighbours.
- Surface-to-volume ratio problem: As the number of cores increases, the amount of computation per core (volume) decreases while the relative amount of communication (surface) increases, eventually dominating the entire computation.
  $\longrightarrow$ We can always do larger simulations, but not smaller simulations faster.



Durham University

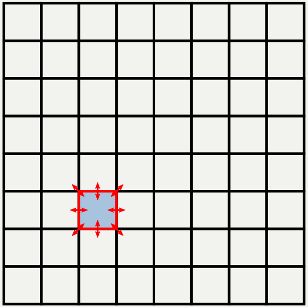
- Distributed-memory parallelism, e.g. using MPI, is based on data decomposition, i.e. each processor is assigned part of the problem to work on and communicates with its neighbours.
- Surface-to-volume ratio problem: As the number of cores increases, the amount of computation per core (volume) decreases while the relative amount of communication (surface) increases, eventually dominating the entire computation.
  $\longrightarrow$ We can always do larger simulations, but not smaller simulations faster.



Task-based parallelism

The problem with distributed-memory parallelism

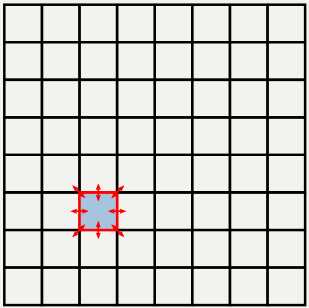
- Distributed-memory parallelism, e.g. using MPI, is based on data decomposition, i.e. each processor is assigned part of the problem to work on and communicates with its neighbours.
- Surface-to-volume ratio problem: As the number of cores increases, the amount of computation per core (volume) decreases while the relative amount of communication (surface) increases, eventually dominating the entire computation.
  $\longrightarrow$ We can always do larger simulations, but not smaller simulations faster.



Task-based parallelism

The problem with distributed-memory parallelism

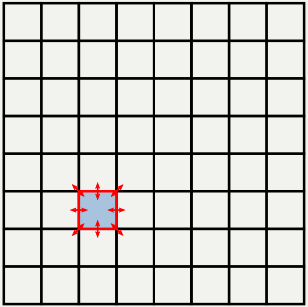
- Distributed-memory parallelism, e.g. using MPI, is based on data decomposition, i.e. each processor is assigned part of the problem to work on and communicates with its neighbours.
- Surface-to-volume ratio problem: As the number of cores increases, the amount of computation per core (volume) decreases while the relative amount of communication (surface) increases, eventually dominating the entire computation.
  $\longrightarrow$ We can always do larger simulations, but not smaller simulations faster.



Task-based parallelism

The problem with distributed-memory parallelism

- Distributed-memory parallelism, e.g. using MPI, is based on data decomposition, i.e. each processor is assigned part of the problem to work on and communicates with its neighbours.
- Surface-to-volume ratio problem: As the number of cores increases, the amount of computation per core (volume) decreases while the relative amount of communication (surface) increases, eventually dominating the entire computation.
  $\longrightarrow$ We can always do larger simulations, but not smaller simulations faster.



Task-based parallelism

The problem with OpenMP

- Shared-memory parallelism using OpenMP, i.e. annotating an inherently serial code, is often hampered by frequent synchronization.
- Concurrency problems need to be addressed explicitly, e.g. using barriers or atomic instructions.
- These overheads associated with these two problems only get worse as the number of cores increases.

```
for ( i = 0 ; i < N ; i++ ) {
    ...
    globalvar += ...
    }
```

Task-based parallelism

The problem with OpenMP

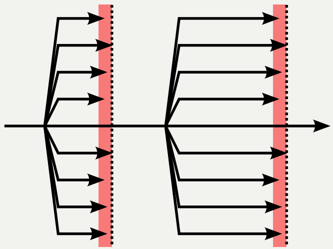
- Shared-memory parallelism using OpenMP, i.e. annotating an inherently serial code, is often hampered by frequent synchronization.
- Concurrency problems need to be addressed explicitly, e.g. using barriers or atomic instructions.
- These overheads associated with these two problems only get worse as the number of cores increases.

```
#pragma omp parallel for
for ( i = 0 ; i < N ; i++ ) {
    ...
    globalvar += ...
    }
```

Task-based parallelism

The problem with OpenMP

- Shared-memory parallelism using OpenMP, i.e. annotating an inherently serial code, is often hampered by frequent synchronization.
- Concurrency problems need to be addressed explicitly, e.g. using barriers or atomic instructions.
- These overheads associated with these two problems only get worse as the number of cores increases.



Task-based parallelism
The problem with OpenMP

- Shared-memory parallelism using OpenMP, i.e. annotating an inherently serial code, is often hampered by frequent synchronization.
- Concurrency problems need to be addressed explicitly, e.g. using barriers or atomic instructions.
- These overheads associated with these two problems only get worse as the number of cores increases.

```
#pragma omp parallel for
for ( i = 0 ; i < N ; i++ ) {
    ...
    #pragma omp critical
    globalvar += ...
    }
```

Task-based parallelism

The problem with OpenMP

- Shared-memory parallelism using OpenMP, i.e. annotating an inherently serial code, is often hampered by frequent synchronization.
- Concurrency problems need to be addressed explicitly, e.g. using barriers or atomic instructions.
- These overheads associated with these two problems only get worse as the number of cores increases.

```
#pragma omp parallel for
for ( i = 0 ; i < N ; i++ ) {
    ...
    #pragma omp critical
    globalvar += ...
    }
```

Task-based parallelism

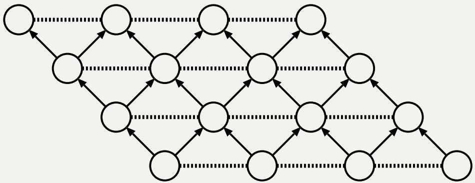
Main concepts

- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.

- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.

- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.

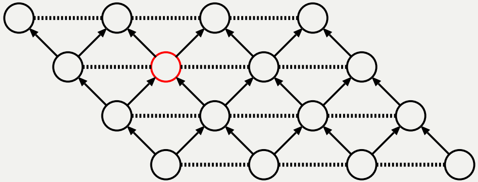
- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  - Which tasks it depends on,
  - Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.



Task-based parallelism

Main concepts

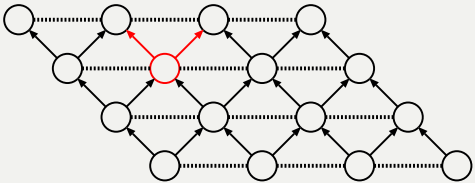
- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  - Which tasks it depends on,
  - Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.



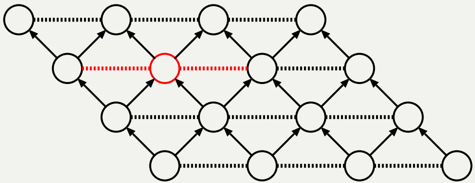
Task-based parallelism

Main concepts

- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  - Which tasks it depends on,
  - Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.



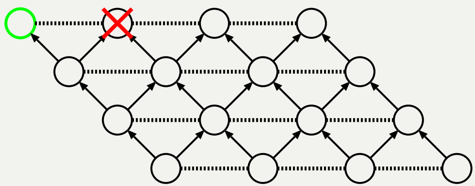
- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  - Which tasks it depends on,
  - Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.



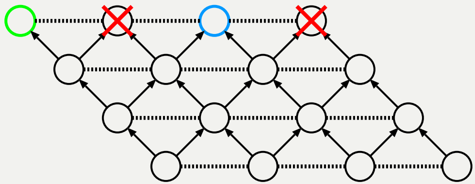
Task-based parallelism

Main concepts

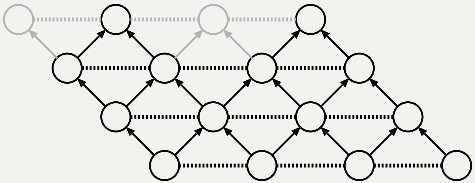
- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  - Which tasks it depends on,
  - Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.



- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  - Which tasks it depends on,
  - Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.



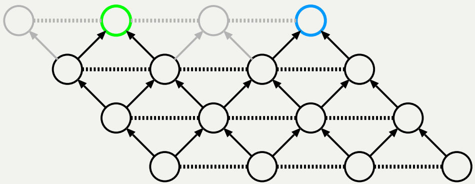
- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  - Which tasks it depends on,
  - Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.



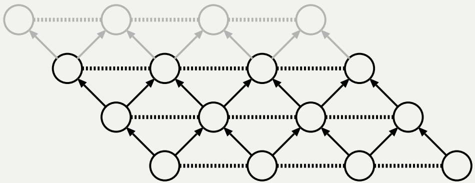
Task-based parallelism

Main concepts

- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  - Which tasks it depends on,
  - Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.



- Shared-memory parallel programming paradigm in which the computation is formulated in an implicitly parallelizable way that automatically avoids most of the problems associated with concurrency and load-balancing.
- We first reduce the problem to a set of inter-dependent tasks.
- For each task, we need to know:
  - Which tasks it depends on,
  - Which tasks it conflicts with.
- Each thread then picks up a task which has no unresolved dependencies or conflicts and computes it.



Task-based parallelism

Main advantages

- The order in which the tasks are processed is highly dynamic and adapts automatically to load imbalances.
- If the dependencies and conflicts are specified correctly, we do not have to worry about concurrency at the level of the individual tasks.
  $\longrightarrow$ No need for expensive explicit locking, synchronization, or atomic operations.
- The same approach can be applied to more unconventional many-core systems such as GPUs or the Intel Phi.
- However, this usually means that we have to completely re-think our entire computation, e.g. redesign it from scratch to make it task-based.

Task-based parallelism

Main advantages

- The order in which the tasks are processed is highly dynamic and adapts automatically to load imbalances.
- If the dependencies and conflicts are specified correctly, we do not have to worry about concurrency at the level of the individual tasks.
  $\longrightarrow$ No need for expensive explicit locking, synchronization, or atomic operations.
- The same approach can be applied to more unconventional many-core systems such as GPUs or the Intel Phi.
- However, this usually means that we have to completely re-think our entire computation, e.g. redesign it from scratch to make it task-based.

Task-based parallelism
Main advantages

- The order in which the tasks are processed is highly dynamic and adapts automatically to load imbalances.
- If the dependencies and conflicts are specified correctly, we do not have to worry about concurrency at the level of the individual tasks.
  $\longrightarrow$ No need for expensive explicit locking, synchronization, or atomic operations.
- The same approach can be applied to more unconventional many-core systems such as GPUs or the Intel Phi.
- However, this usually means that we have to completely re-think our entire computation, e.g. redesign it from scratch to make it task-based.

- The order in which the tasks are processed is highly dynamic and adapts automatically to load imbalances.
- If the dependencies and conflicts are specified correctly, we do not have to worry about concurrency at the level of the individual tasks.
  $\longrightarrow$ No need for expensive explicit locking, synchronization, or atomic operations.
- The same approach can be applied to more unconventional many-core systems such as GPUs or the Intel Phi.
- However, this usually means that we have to completely re-think our entire computation, e.g. redesign it from scratch to make it task-based.

Task-based parallelism

Main advantages

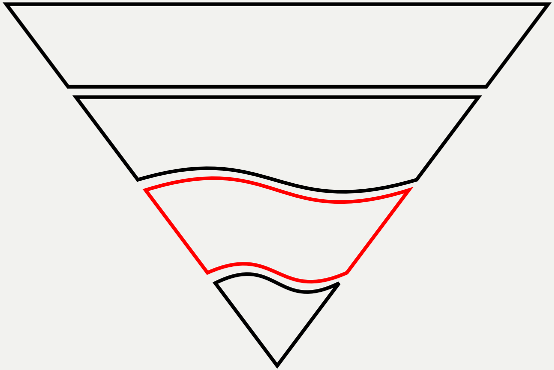
- The order in which the tasks are processed is highly dynamic and adapts automatically to load imbalances.
- If the dependencies and conflicts are specified correctly, we do not have to worry about concurrency at the level of the individual tasks.
  $\longrightarrow$ No need for expensive explicit locking, synchronization, or atomic operations.
- The same approach can be applied to more unconventional many-core systems such as GPUs or the Intel Phi.
- However, this usually means that we have to completely re-think our entire computation, e.g. redesign it from scratch to make it task-based.

Domain Science

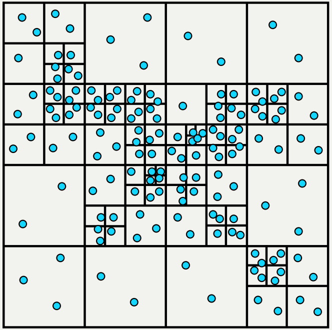
Software

Algorithms

Paradigms

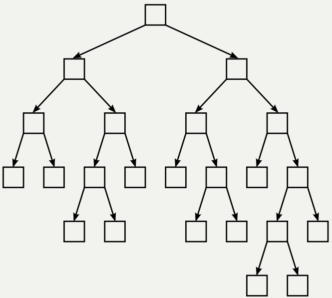


- Spatial trees are the most commonly used approach to neighbour-finding, as the particle distribution can be irregular.
- Neighbour-finding up and down the tree is simple, but has some problems:
  - Worst-case cost in $\mathcal{O}(N^{2/3})$ per particle.
  - Low cache efficiency due to scattered memory access.
  - Symmetries cannot be exploited, i.e. each particle pair is found twice.
- Parallelization is trivial, but only because symmetries are not exploited.



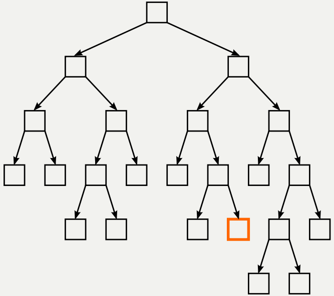
Algorithms for SPH
Neighbour-finding with trees

- Spatial trees are the most commonly used approach to neighbour-finding, as the particle distribution can be irregular.
- Neighbour-finding up and down the tree is simple, but has some problems:
  - Worst-case cost in $\mathcal{O}(N^{2/3})$ per particle.
  - Low cache efficiency due to scattered memory access.
  - Symmetries cannot be exploited, i.e. each particle pair is found twice.
- Parallelization is trivial, but only because symmetries are not exploited.



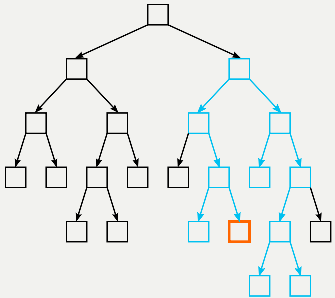
Algorithms for SPH
Neighbour-finding with trees

- Spatial trees are the most commonly used approach to neighbour-finding, as the particle distribution can be irregular.
- Neighbour-finding up and down the tree is simple, but has some problems:
  - Worst-case cost in $\mathcal{O}(N^{2/3})$ per particle.
  - Low cache efficiency due to scattered memory access.
  - Symmetries cannot be exploited, i.e. each particle pair is found twice.
- Parallelization is trivial, but only because symmetries are not exploited.

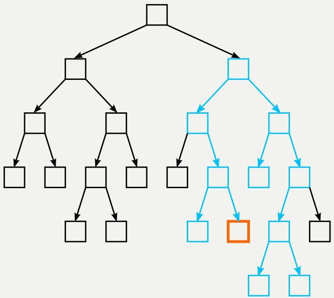


Algorithms for SPH
Neighbour-finding with trees

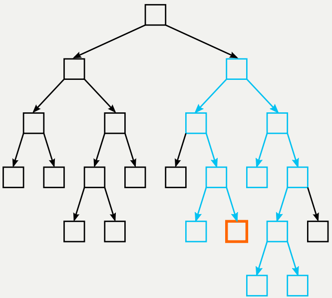
- Spatial trees are the most commonly used approach to neighbour-finding, as the particle distribution can be irregular.
- Neighbour-finding up and down the tree is simple, but has some problems:
  - Worst-case cost in $\mathcal{O}(N^{2/3})$ per particle.
  - Low cache efficiency due to scattered memory access.
  - Symmetries cannot be exploited, i.e. each particle pair is found twice.
- Parallelization is trivial, but only because symmetries are not exploited.



- Spatial trees are the most commonly used approach to neighbour-finding, as the particle distribution can be irregular.
- Neighbour-finding up and down the tree is simple, but has some problems:
  - Worst-case cost in $\mathcal{O}(N^{2/3})$ per particle.
  - Low cache efficiency due to scattered memory access.
  - Symmetries cannot be exploited, i.e. each particle pair is found twice.
- Parallelization is trivial, but only because symmetries are not exploited.



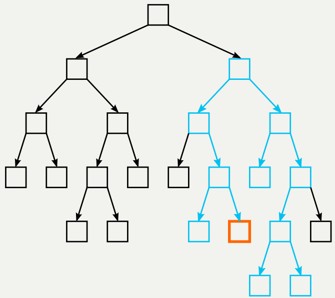
- Spatial trees are the most commonly used approach to neighbour-finding, as the particle distribution can be irregular.
- Neighbour-finding up and down the tree is simple, but has some problems:
  - Worst-case cost in $\mathcal{O}(N^{2/3})$ per particle.
  - Low cache efficiency due to scattered memory access.
  - Symmetries cannot be exploited, i.e. each particle pair is found twice.
- Parallelization is trivial, but only because symmetries are not exploited.



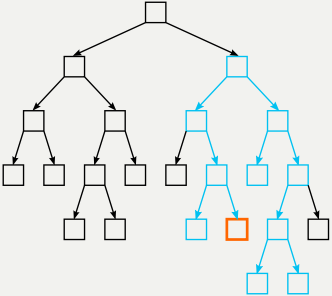
Algorithms for SPH

Neighbour-finding with trees

- Spatial trees are the most commonly used approach to neighbour-finding, as the particle distribution can be irregular.
- Neighbour-finding up and down the tree is simple, but has some problems:
  - Worst-case cost in $\mathcal{O}(N^{2/3})$ per particle.
  - Low cache efficiency due to scattered memory access.
  - Symmetries cannot be exploited, i.e. each particle pair is found twice.
- Parallelization is trivial, but only because symmetries are not exploited.



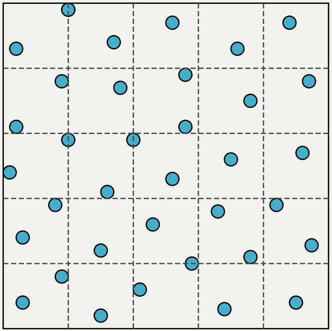
- Spatial trees are the most commonly used approach to neighbour-finding, as the particle distribution can be irregular.
- Neighbour-finding up and down the tree is simple, but has some problems:
  - Worst-case cost in $\mathcal{O}(N^{2/3})$ per particle.
  - Low cache efficiency due to scattered memory access.
  - Symmetries cannot be exploited, i.e. each particle pair is found twice.
- Parallelization is trivial, but only because symmetries are not exploited.



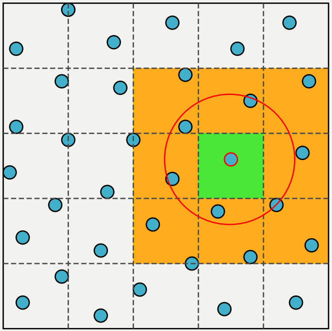
Algorithms for SPH

Hierarchical cell pairs

- We start by splitting the simulation domain into rectangular cells of edge length at least h_{max}.
- All interacting particle pairs are then in either in the same cell, or in a pair of neighbouring cells.
- Finding all neighbours within each cell or between each pair of cells can be used as a task.
- If the particles in the cell or cell pair are sufficiently small, the task can be split.
- Finally, the particles in each cell pair are first sorted along the cell pair axis to speed-up neighbour-finding.



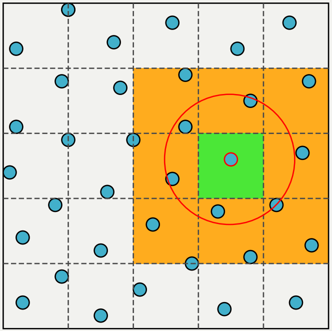
- We start by splitting the simulation domain into rectangular cells of edge length at least $h_{\max}$.
- All interacting particle pairs are then in either in the same cell, or in a pair of neighbouring cells.
- Finding all neighbours within each cell or between each pair of cells can be used as a task.
- If the particles in the cell or cell pair are sufficiently small, the task can be split.
- Finally, the particles in each cell pair are first sorted along the cell pair axis to speed-up neighbour-finding.



Algorithms for SPH

Hierarchical cell pairs

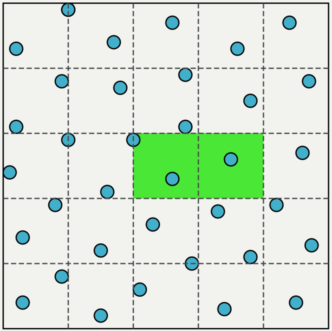
- We start by splitting the simulation domain into rectangular cells of edge length at least h_{max}.
- All interacting particle pairs are then in either in the same cell, or in a pair of neighbouring cells.
- Finding all neighbours within each cell or between each pair of cells can be used as a task.
- If the particles in the cell or cell pair are sufficiently small, the task can be split.
- Finally, the particles in each cell pair are first sorted along the cell pair axis to speed-up neighbour-finding.



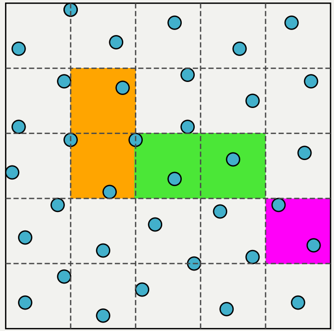
Algorithms for SPH

Hierarchical cell pairs

- We start by splitting the simulation domain into rectangular cells of edge length at least h_{max}.
- All interacting particle pairs are then in either in the same cell, or in a pair of neighbouring cells.
- Finding all neighbours within each cell or between each pair of cells can be used as a task.
- If the particles in the cell or cell pair are sufficiently small, the task can be split.
- Finally, the particles in each cell pair are first sorted along the cell pair axis to speed-up neighbour-finding.



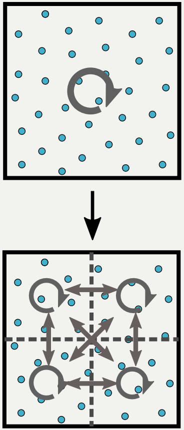
- We start by splitting the simulation domain into rectangular cells of edge length at least h_{max}.
- All interacting particle pairs are then in either in the same cell, or in a pair of neighbouring cells.
- Finding all neighbours within each cell or between each pair of cells can be used as a task.
- If the particles in the cell or cell pair are sufficiently small, the task can be split.
- Finally, the particles in each cell pair are first sorted along the cell pair axis to speed-up neighbour-finding.



Algorithms for SPH

Hierarchical cell pairs

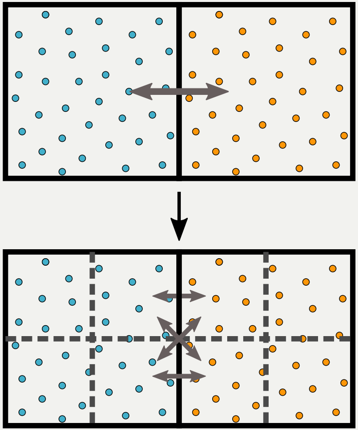
- We start by splitting the simulation domain into rectangular cells of edge length at least h_{max}.
- All interacting particle pairs are then in either in the same cell, or in a pair of neighbouring cells.
- Finding all neighbours within each cell or between each pair of cells can be used as a task.
- If the particles in the cell or cell pair are sufficiently small, the task can be split.
- Finally, the particles in each cell pair are first sorted along the cell pair axis to speed-up neighbour-finding.



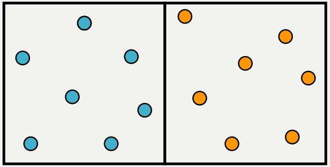
Algorithms for SPH

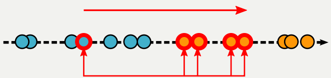
Hierarchical cell pairs

- We start by splitting the simulation domain into rectangular cells of edge length at least h_{max}.
- All interacting particle pairs are then in either in the same cell, or in a pair of neighbouring cells.
- Finding all neighbours within each cell or between each pair of cells can be used as a task.
- If the particles in the cell or cell pair are sufficiently small, the task can be split.
- Finally, the particles in each cell pair are first sorted along the cell pair axis to speed-up neighbour-finding.



- We start by splitting the simulation domain into rectangular cells of edge length at least h_{max}.
- All interacting particle pairs are then in either in the same cell, or in a pair of neighbouring cells.
- Finding all neighbours within each cell or between each pair of cells can be used as a task.
- If the particles in the cell or cell pair are sufficiently small, the task can be split.
- Finally, the particles in each cell pair are first sorted along the cell pair axis to speed-up neighbour-finding.

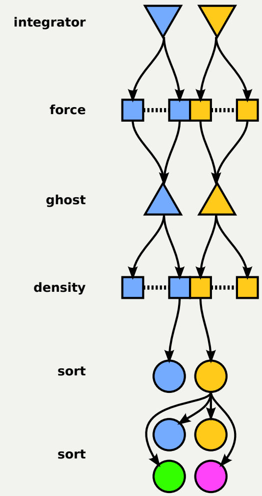




Algorithms for SPH

Task hierarchy

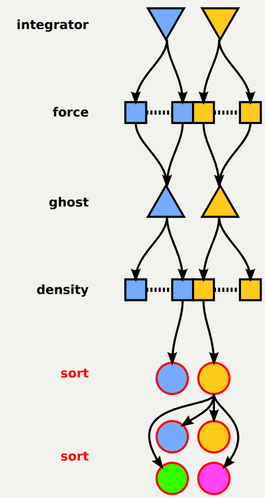
- Three main task types: Sorting, self-interactions, and pair-interactions.
- "Ghost" tasks are added to group dependencies between the density and force tasks of each cell.
- Each pair-interaction task depends on the sort tasks of the cells involved.
- Each sorting task depends on the sorting tasks of its sub-cells (merge-sort).
- Tasks on overlapping cells conflict, i.e. they can not execute concurrently.
- Finally, integrator tasks for each cell depend on the forces having been computed.



Algorithms for SPH

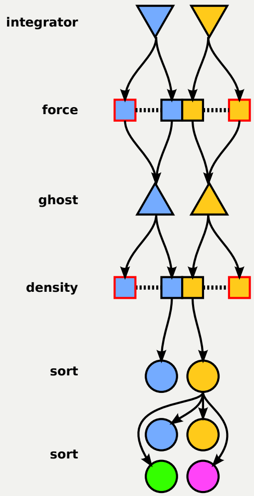
Task hierarchy

- Three main task types: Sorting, self-interactions, and pair-interactions.
- "Ghost" tasks are added to group dependencies between the density and force tasks of each cell.
- Each pair-interaction task depends on the sort tasks of the cells involved.
- Each sorting task depends on the sorting tasks of its sub-cells (merge-sort).
- Tasks on overlapping cells conflict, i.e. they can not execute concurrently.
- Finally, integrator tasks for each cell depend on the forces having been computed.



Algorithms for SPH
Task hierarchy

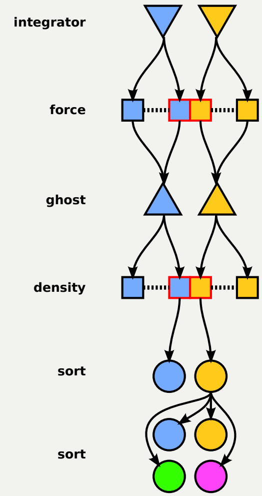
- Three main task types: Sorting, self-interactions, and pair-interactions.
- “Ghost” tasks are added to group dependencies between the density and force tasks of each cell.
- Each pair-interaction task depends on the sort tasks of the cells involved.
- Each sorting task depends on the sorting tasks of its sub-cells (merge-sort).
- Tasks on overlapping cells conflict, i.e. they can not execute concurrently.
- Finally, integrator tasks for each cell depend on the forces having been computed.



Algorithms for SPH

Task hierarchy

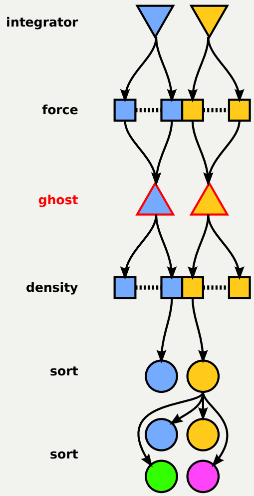
- Three main task types: Sorting, self-interactions, and pair-interactions.
- “Ghost” tasks are added to group dependencies between the density and force tasks of each cell.
- Each pair-interaction task depends on the sort tasks of the cells involved.
- Each sorting task depends on the sorting tasks of its sub-cells (merge-sort).
- Tasks on overlapping cells conflict, i.e. they can not execute concurrently.
- Finally, integrator tasks for each cell depend on the forces having been computed.



Algorithms for SPH

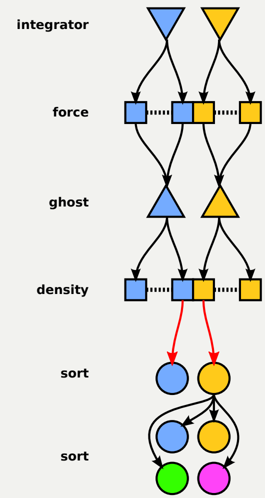
Task hierarchy

- Three main task types: Sorting, self-interactions, and pair-interactions.
- "Ghost" tasks are added to group dependencies between the density and force tasks of each cell.
- Each pair-interaction task depends on the sort tasks of the cells involved.
- Each sorting task depends on the sorting tasks of its sub-cells (merge-sort).
- Tasks on overlapping cells conflict, i.e. they can not execute concurrently.
- Finally, integrator tasks for each cell depend on the forces having been computed.



Algorithms for SPH
Task hierarchy

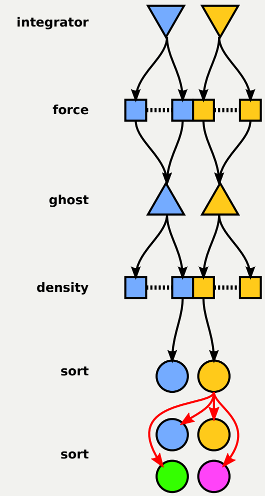
- Three main task types: Sorting, self-interactions, and pair-interactions.
- "Ghost" tasks are added to group dependencies between the density and force tasks of each cell.
- Each pair-interaction task depends on the sort tasks of the cells involved.
- Each sorting task depends on the sorting tasks of its sub-cells (merge-sort).
- Tasks on overlapping cells conflict, i.e. they can not execute concurrently.
- Finally, integrator tasks for each cell depend on the forces having been computed.



Algorithms for SPH

Task hierarchy

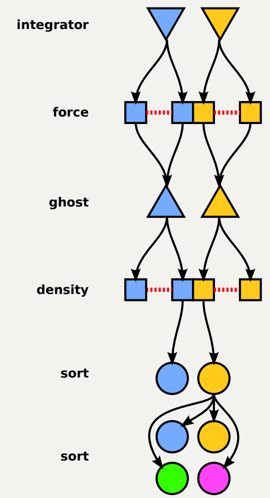
- Three main task types: Sorting, self-interactions, and pair-interactions.
- "Ghost" tasks are added to group dependencies between the density and force tasks of each cell.
- Each pair-interaction task depends on the sort tasks of the cells involved.
- Each sorting task depends on the sorting tasks of its sub-cells (merge-sort).
- Tasks on overlapping cells conflict, i.e. they can not execute concurrently.
- Finally, integrator tasks for each cell depend on the forces having been computed.



Algorithms for SPH

Task hierarchy

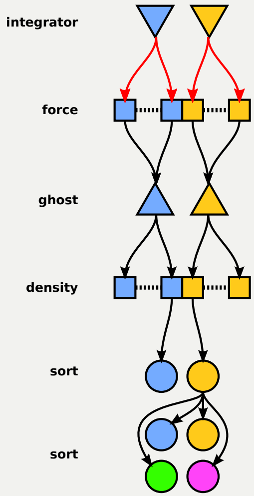
- Three main task types: Sorting, self-interactions, and pair-interactions.
- "Ghost" tasks are added to group dependencies between the density and force tasks of each cell.
- Each pair-interaction task depends on the sort tasks of the cells involved.
- Each sorting task depends on the sorting tasks of its sub-cells (merge-sort).
- Tasks on overlapping cells conflict, i.e. they can not execute concurrently.
- Finally, integrator tasks for each cell depend on the forces having been computed.

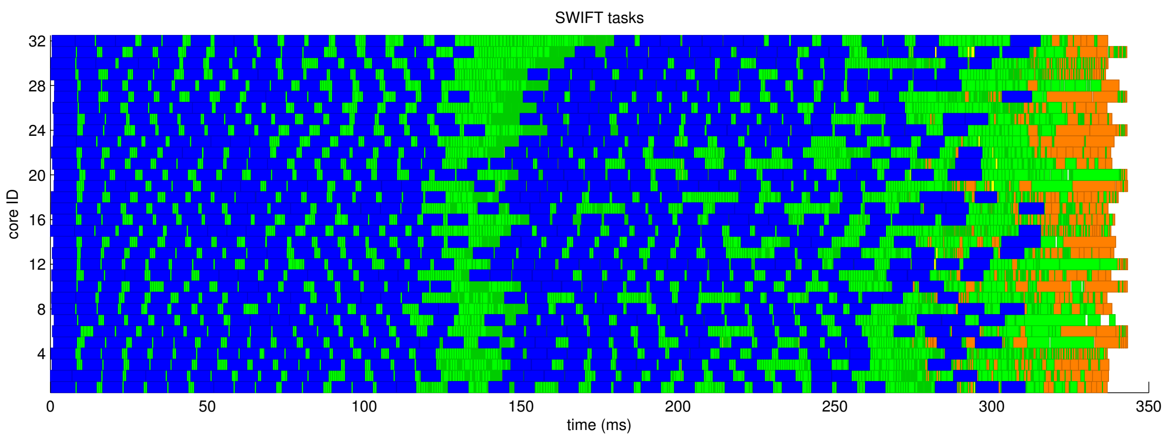


Algorithms for SPH

Task hierarchy

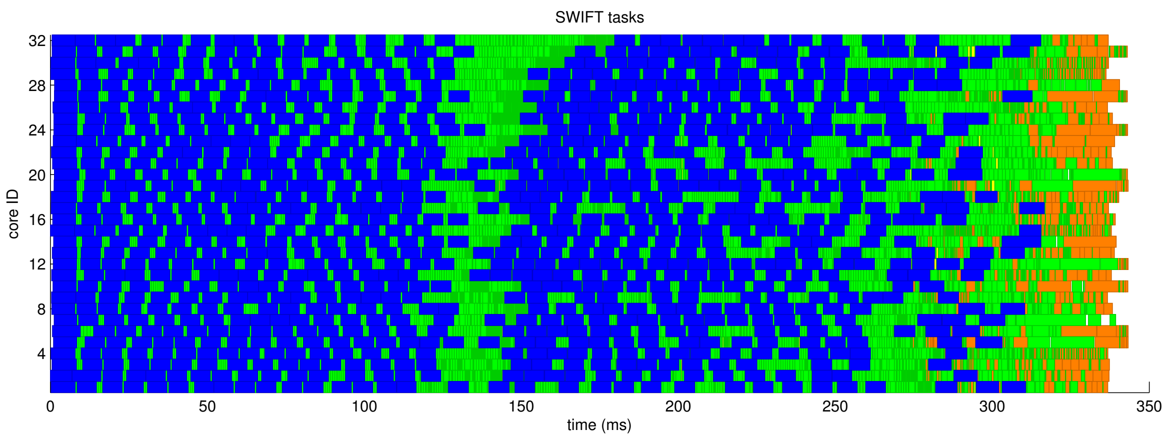
- Three main task types: Sorting, self-interactions, and pair-interactions.
- “Ghost” tasks are added to group dependencies between the density and force tasks of each cell.
- Each pair-interaction task depends on the sort tasks of the cells involved.
- Each sorting task depends on the sorting tasks of its sub-cells (merge-sort).
- Tasks on overlapping cells conflict, i.e. they can not execute concurrently.
- Finally, integrator tasks for each cell depend on the forces having been computed.





- Each core has it's own task queue and uses work-stealing when empty.
- Each core has a preference for tasks involving cells which were used previously to improve cache re-use.

Algorithms for SPH
Dynamic task allocation

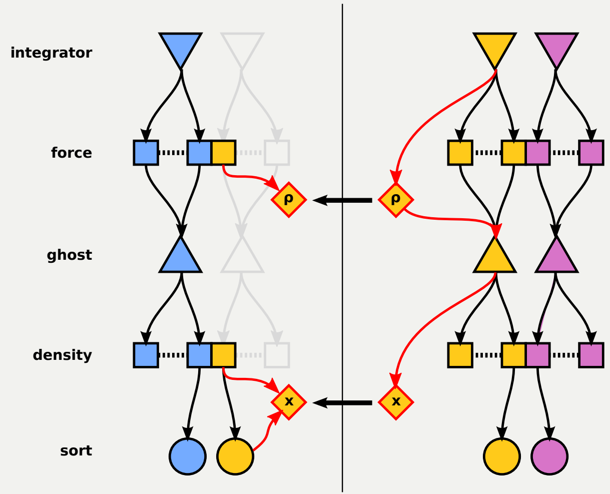


- Each core has it's own task queue and uses work-stealing when empty.
- Each core has a preference for tasks involving cells which were used previously to improve cache re-use.

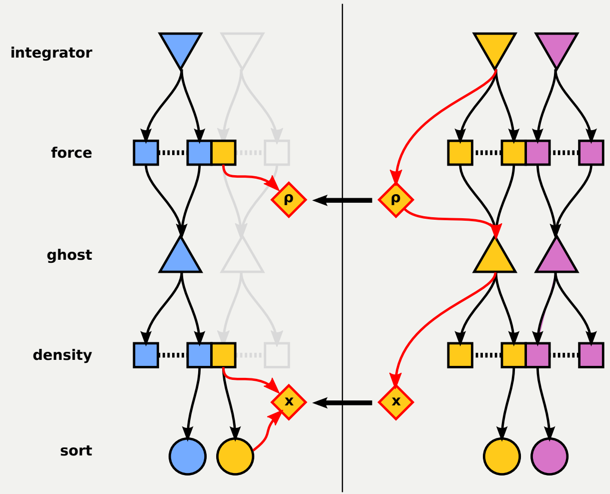
Algorithms for SPH

Hybrid shared/distributed-memory parallelism

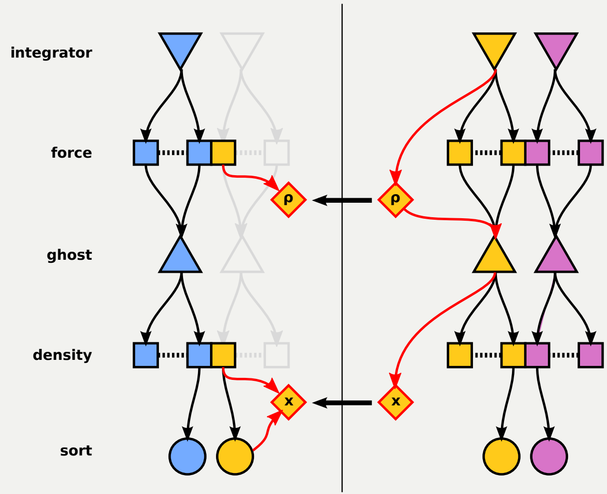
- Domain decomposition between multi-core nodes, task-based parallelism within each node.
- Each cell is owned by one node alone, and tasks involving foreign nodes are made dependent on communication tasks.
- Communication tasks execute dynamically and asynchronously, overlapping with computation.



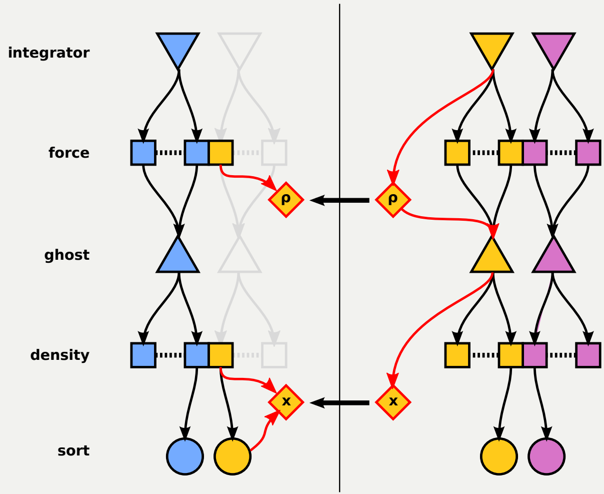
- Domain decomposition between multi-core nodes, task-based parallelism within each node.
- Each cell is owned by one node alone, and tasks involving foreign nodes are made dependent on communication tasks.
- Communication tasks execute dynamically and asynchronously, overlapping with computation.



- Domain decomposition between multi-core nodes, task-based parallelism within each node.
- Each cell is owned by one node alone, and tasks involving foreign nodes are made dependent on communication tasks.
- Communication tasks execute dynamically and asynchronously, overlapping with computation.



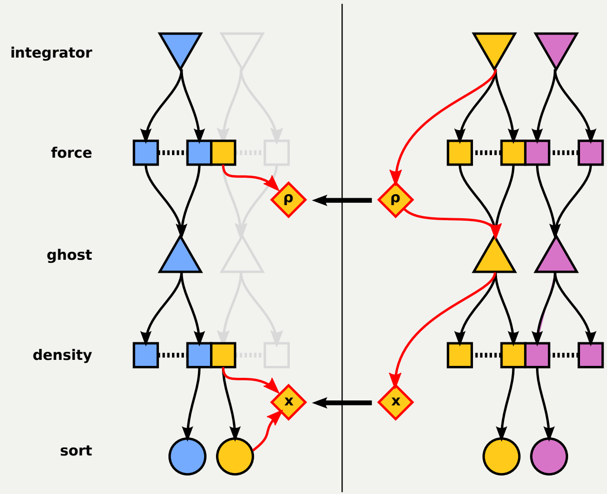
- Domain decomposition between multi-core nodes, task-based parallelism within each node.
- Each cell is owned by one node alone, and tasks involving foreign nodes are made dependent on communication tasks.
- Communication tasks execute dynamically and asynchronously, overlapping with computation.



Algorithms for SPH

Hybrid shared/distributed-memory parallelism

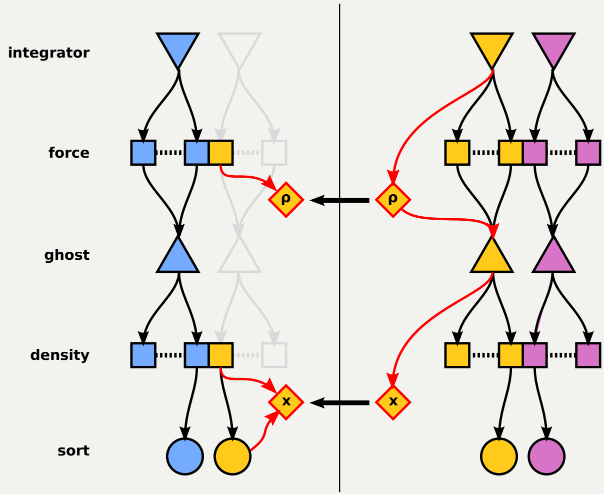
- Domain decomposition between multi-core nodes, task-based parallelism within each node.
- Each cell is owned by one node alone, and tasks involving foreign nodes are made dependent on communication tasks.
- Communication tasks execute dynamically and asynchronously, overlapping with computation.



Algorithms for SPH

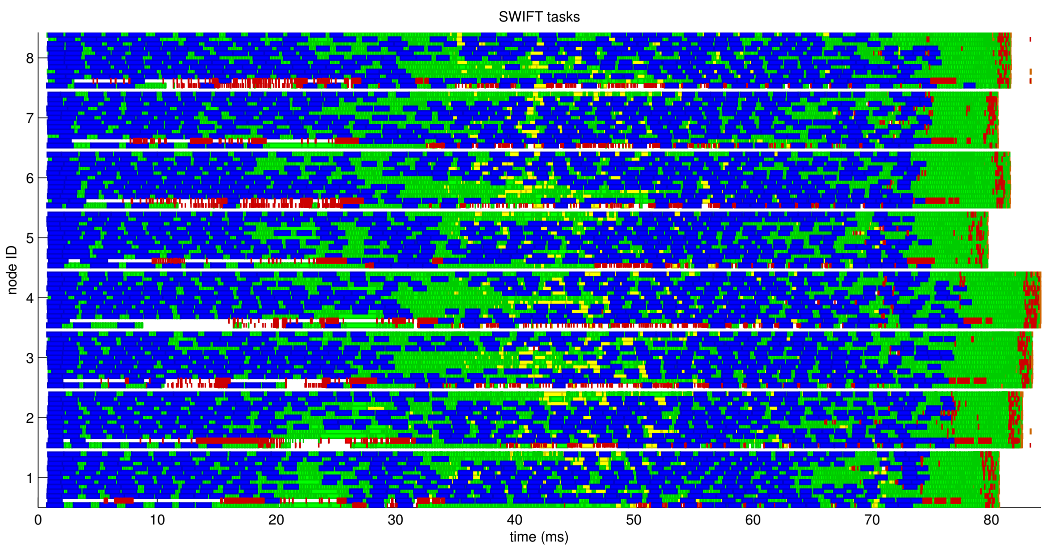
Hybrid shared/distributed-memory parallelism

- Domain decomposition between multi-core nodes, task-based parallelism within each node.
- Each cell is owned by one node alone, and tasks involving foreign nodes are made dependent on communication tasks.
- Communication tasks execute dynamically and asynchronously, overlapping with computation.



Algorithms for SPH

Hybrid shared/distributed-memory parallelism



- This approach still suffers the same surface-to-volume ratio problem as other distributed-memory computations, but at a much smaller scale.
  $\longrightarrow$ Only one MPI-node per physical node.
- Future increases in the number of cores per node will not affect distributed-memory parallel scalability.
- The domain decomposition can be computed along the task graph with standard graph partitioning, e.g. METIS.
  $\longrightarrow$ Splitting the actual *work*, and not just the *data*.

- This approach still suffers the same surface-to-volume ratio problem as other distributed-memory computations, but at a much smaller scale.
  $\longrightarrow$ Only one MPI-node per physical node.
- Future increases in the number of cores per node will not affect distributed-memory parallel scalability.
- The domain decomposition can be computed along the task graph with standard graph partitioning, e.g. METIS.
  $\longrightarrow$ Splitting the actual *work*, and not just the *data*.

Algorithms for SPH

Hybrid shared/distributed-memory parallelism

- This approach still suffers the same surface-to-volume ratio problem as other distributed-memory computations, but at a much smaller scale.
  $\longrightarrow$ Only one MPI-node per physical node.
- Future increases in the number of cores per node will not affect distributed-memory parallel scalability.
- The domain decomposition can be computed along the task graph with standard graph partitioning, e.g. METIS.
  $\longrightarrow$ Splitting the actual *work*, and not just the *data*.

Algorithms for SPH

Hybrid shared/distributed-memory parallelism

- This approach still suffers the same surface-to-volume ratio problem as other distributed-memory computations, but at a much smaller scale.
  $\longrightarrow$ Only one MPI-node per physical node.
- Future increases in the number of cores per node will not affect distributed-memory parallel scalability.
- The domain decomposition can be computed along the task graph with standard graph partitioning, e.g. METIS.
  $\longrightarrow$ Splitting the actual *work*, and not just the *data*.

- This approach still suffers the same surface-to-volume ratio problem as other distributed-memory computations, but at a much smaller scale.
  $\longrightarrow$ Only one MPI-node per physical node.
- Future increases in the number of cores per node will not affect distributed-memory parallel scalability.
- The domain decomposition can be computed along the task graph with standard graph partitioning, e.g. METIS.
  $\longrightarrow$ Splitting the actual *work*, and not just the *data*.

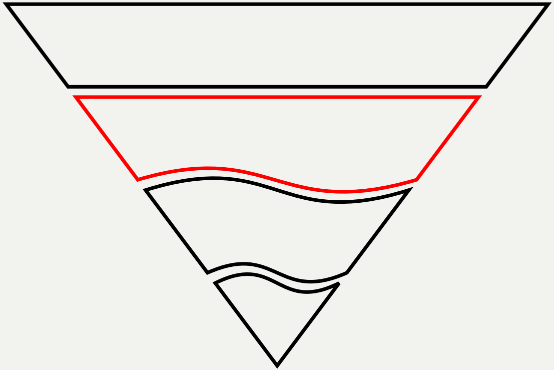
- This approach still suffers the same surface-to-volume ratio problem as other distributed-memory computations, but at a much smaller scale.
  $\longrightarrow$ Only one MPI-node per physical node.
- Future increases in the number of cores per node will not affect distributed-memory parallel scalability.
- The domain decomposition can be computed along the task graph with standard graph partitioning, e.g. METIS.
  $\longrightarrow$ Splitting the actual *work*, and not just the *data*.

Domain Science

Software

Algorithms

Paradigms



- Close collaboration with the Institute for Computational Cosmology (ICC) at Durham University.
  $\longrightarrow$ Make sure we're building a software that can actually be used.
- Main goal is to replace GADGET, the most popular Open-Source cosmological simulation code.
- Massively multi-scale problems, with millions to billions of particles, run on both desktops and supercomputers.
  $\longrightarrow$ Include support for GPUs in order to take some of the moderate simulations off the cluster and onto desktop workstations.

Durham University

- Close collaboration with the Institute for Computational Cosmology (ICC) at Durham University.
  $\longrightarrow$ Make sure we're building a software that can actually be used.
- Main goal is to replace GADGET, the most popular Open-Source cosmological simulation code.
- Massively multi-scale problems, with millions to billions of particles, run on both desktops and supercomputers.
  $\longrightarrow$ Include support for GPUs in order to take some of the moderate simulations off the cluster and onto desktop workstations.

- Close collaboration with the Institute for Computational Cosmology (ICC) at Durham University.
  $\longrightarrow$ Make sure we're building a software that can actually be used.
- Main goal is to replace GADGET, the most popular Open-Source cosmological simulation code.
- Massively multi-scale problems, with millions to billions of particles, run on both desktops and supercomputers.
  $\longrightarrow$ Include support for GPUs in order to take some of the moderate simulations off the cluster and onto desktop workstations.

- Close collaboration with the Institute for Computational Cosmology (ICC) at Durham University.
  $\longrightarrow$ Make sure we're building a software that can actually be used.
- Main goal is to replace GADGET, the most popular Open-Source cosmological simulation code.
- Massively multi-scale problems, with millions to billions of particles, run on both desktops and supercomputers.
  $\longrightarrow$ Include support for GPUs in order to take some of the moderate simulations off the cluster and onto desktop workstations.

- Close collaboration with the Institute for Computational Cosmology (ICC) at Durham University.
  $\longrightarrow$ Make sure we're building a software that can actually be used.
- Main goal is to replace GADGET, the most popular Open-Source cosmological simulation code.
- Massively multi-scale problems, with millions to billions of particles, run on both desktops and supercomputers.
  $\longrightarrow$ Include support for GPUs in order to take some of the moderate simulations off the cluster and onto desktop workstations.

- Close collaboration with the Institute for Computational Cosmology (ICC) at Durham University.
  $\longrightarrow$ Make sure we're building a software that can actually be used.
- Main goal is to replace GADGET, the most popular Open-Source cosmological simulation code.
- Massively multi-scale problems, with millions to billions of particles, run on both desktops and supercomputers.
  $\longrightarrow$ Include support for GPUs in order to take some of the moderate simulations off the cluster and onto desktop workstations.

- Close collaboration with the Institute for Computational Cosmology (ICC) at Durham University.
  $\longrightarrow$ Make sure we're building a software that can actually be used.
- Main goal is to replace GADGET, the most popular Open-Source cosmological simulation code.
- Massively multi-scale problems, with millions to billions of particles, run on both desktops and supercomputers.
  $\longrightarrow$ Include support for GPUs in order to take some of the moderate simulations off the cluster and onto desktop workstations.

- Close collaboration with the Institute for Computational Cosmology (ICC) at Durham University.
  $\longrightarrow$ Make sure we're building a software that can actually be used.
- Main goal is to replace GADGET, the most popular Open-Source cosmological simulation code.
- Massively multi-scale problems, with millions to billions of particles, run on both desktops and supercomputers.
  $\longrightarrow$ Include support for GPUs in order to take some of the moderate simulations off the cluster and onto desktop workstations.

- 15'000 lines of C written by *Computer Scientists*, using `pthreads` and MPI, fully Open-Source.
- Explicit vectorization (switched off in the benchmarks here for fairness).
- Mixed-precision arithmetic for better performance and vectorization, lower memory requirements.
- Currently porting to CUDA-based GPUs and the Intel Phi.
- Easily extensible for different physics, kernel functions, setups, etc...
  $\longrightarrow$ Code within the tasks is not parallel-aware, so no specific know-how needed to extend it.

- 15'000 lines of C written by *Computer Scientists*, using `pthreads` and MPI, fully Open-Source.
- Explicit vectorization (switched off in the benchmarks here for fairness).
- Mixed-precision arithmetic for better performance and vectorization, lower memory requirements.
- Currently porting to CUDA-based GPUs and the Intel Phi.
- Easily extensible for different physics, kernel functions, setups, etc...
  $\longrightarrow$ Code within the tasks is not parallel-aware, so no specific know-how needed to extend it.

- 15'000 lines of C written by *Computer Scientists*, using `pthreads` and MPI, fully Open-Source.
- Explicit vectorization (switched off in the benchmarks here for fairness).
- Mixed-precision arithmetic for better performance and vectorization, lower memory requirements.
- Currently porting to CUDA-based GPUs and the Intel Phi.
- Easily extensible for different physics, kernel functions, setups, etc...
  $\longrightarrow$ Code within the tasks is not parallel-aware, so no specific know-how needed to extend it.

Durham University

- 15'000 lines of C written by *Computer Scientists*, using `pthreads` and MPI, fully Open-Source.
- Explicit vectorization (switched off in the benchmarks here for fairness).
- Mixed-precision arithmetic for better performance and vectorization, lower memory requirements.
- Currently porting to CUDA-based GPUs and the Intel Phi.
- Easily extensible for different physics, kernel functions, setups, etc...
  $\longrightarrow$ Code within the tasks is not parallel-aware, so no specific know-how needed to extend it.

- 15'000 lines of C written by *Computer Scientists*, using `pthreads` and MPI, fully Open-Source.
- Explicit vectorization (switched off in the benchmarks here for fairness).
- Mixed-precision arithmetic for better performance and vectorization, lower memory requirements.
- Currently porting to CUDA-based GPUs and the Intel Phi.
- Easily extensible for different physics, kernel functions, setups, etc...
  $\longrightarrow$ Code within the tasks is not parallel-aware, so no specific know-how needed to extend it.

SWIFT
Main features

- 15'000 lines of C written by *Computer Scientists*, using `pthreads` and MPI, fully Open-Source.
- Explicit vectorization (switched off in the benchmarks here for fairness).
- Mixed-precision arithmetic for better performance and vectorization, lower memory requirements.
- Currently porting to CUDA-based GPUs and the Intel Phi.
- Easily extensible for different physics, kernel functions, setups, etc. . .
  $\longrightarrow$ Code within the tasks is not parallel-aware, so no specific know-how needed to extend it.

SWIFT
Main features

- 15'000 lines of C written by *Computer Scientists*, using `pthreads` and MPI, fully Open-Source.
- Explicit vectorization (switched off in the benchmarks here for fairness).
- Mixed-precision arithmetic for better performance and vectorization, lower memory requirements.
- Currently porting to CUDA-based GPUs and the Intel Phi.
- Easily extensible for different physics, kernel functions, setups, etc...
  $\longrightarrow$ Code within the tasks is not parallel-aware, so no specific know-how needed to extend it.

SWIFT

Main features

- 15'000 lines of C written by *Computer Scientists*, using `pthreads` and MPI, fully Open-Source.
- Explicit vectorization (switched off in the benchmarks here for fairness).
- Mixed-precision arithmetic for better performance and vectorization, lower memory requirements.
- Currently porting to CUDA-based GPUs and the Intel Phi.
- Easily extensible for different physics, kernel functions, setups, etc...
  $\longrightarrow$ Code within the tasks is not parallel-aware, so no specific know-how needed to extend it.

Conclusions

Our contributions

- Extended task-based parallelism conceptually to include *task conflicts*.
  $\longrightarrow$ Scheduling is trickier, but much more flexible.
- Faster and more cache-efficient algorithms for *neighbour-finding* in multi-scale particle systems.
  $\longrightarrow$ More than $7\times$ faster than GADGET-2 on a single core.
- Fully data-driven, asynchronous, and dynamic *hybrid shared/distributed-memory parallel model*.
  $\longrightarrow$ Good scaling even for small simulations on large numbers of cores.
- Domain decomposition based on *decomposing the task graph*.
  $\longrightarrow$ Decomposing the actual work, as opposed to just the data.

Conclusions
Our contributions

- Extended task-based parallelism conceptually to include *task conflicts*.
  $\longrightarrow$ Scheduling is trickier, but much more flexible.
- Faster and more cache-efficient algorithms for *neighbour-finding* in multi-scale particle systems.
  $\longrightarrow$ More than $7\times$ faster than GADGET-2 on a single core.
- Fully data-driven, asynchronous, and dynamic *hybrid shared/distributed-memory parallel model*.
  $\longrightarrow$ Good scaling even for small simulations on large numbers of cores.
- Domain decomposition based on *decomposing the task graph*.
  $\longrightarrow$ Decomposing the actual work, as opposed to just the data.

Conclusions

Our contributions

- Extended task-based parallelism conceptually to include *task conflicts*.
  $\longrightarrow$ Scheduling is trickier, but much more flexible.
- Faster and more cache-efficient algorithms for *neighbour-finding* in multi-scale particle systems.
  $\longrightarrow$ More than 7× faster than GADGET-2 on a single core.
- Fully data-driven, asynchronous, and dynamic *hybrid shared/distributed-memory parallel model*.
  $\longrightarrow$ Good scaling even for small simulations on large numbers of cores.
- Domain decomposition based on *decomposing the task graph*.
  $\longrightarrow$ Decomposing the actual work, as opposed to just the data.

Conclusions

Our contributions

- Extended task-based parallelism conceptually to include *task conflicts*.
  $\longrightarrow$ Scheduling is trickier, but much more flexible.
- Faster and more cache-efficient algorithms for *neighbour-finding* in multi-scale particle systems.
  $\longrightarrow$ More than 7× faster than GADGET-2 on a single core.
- Fully data-driven, asynchronous, and dynamic *hybrid shared/distributed-memory parallel model*.
  $\longrightarrow$ Good scaling even for small simulations on large numbers of cores.
- Domain decomposition based on *decomposing the task graph*.
  $\longrightarrow$ Decomposing the actual work, as opposed to just the data.

Conclusions

Our contributions

- Extended task-based parallelism conceptually to include *task conflicts*.
  $\longrightarrow$ Scheduling is trickier, but much more flexible.
- Faster and more cache-efficient algorithms for *neighbour-finding* in multi-scale particle systems.
  $\longrightarrow$ More than $7\times$ faster than GADGET-2 on a single core.
- Fully data-driven, asynchronous, and dynamic *hybrid shared/distributed-memory parallel model*.
  $\longrightarrow$ Good scaling even for small simulations on large numbers of cores.
- Domain decomposition based on *decomposing the task graph*.
  $\longrightarrow$ Decomposing the actual work, as opposed to just the data.

Conclusions

Our contributions

- Extended task-based parallelism conceptually to include *task conflicts*.
  $\longrightarrow$ Scheduling is trickier, but much more flexible.
- Faster and more cache-efficient algorithms for *neighbour-finding* in multi-scale particle systems.
  $\longrightarrow$ More than $7\times$ faster than GADGET-2 on a single core.
- Fully data-driven, asynchronous, and dynamic *hybrid shared/distributed-memory parallel model*.
  $\longrightarrow$ Good scaling even for small simulations on large numbers of cores.
- Domain decomposition based on *decomposing the task graph*.
  $\longrightarrow$ Decomposing the actual work, as opposed to just the data.

Conclusions

Our contributions

- Extended task-based parallelism conceptually to include *task conflicts*.
  $\longrightarrow$ Scheduling is trickier, but much more flexible.
- Faster and more cache-efficient algorithms for *neighbour-finding* in multi-scale particle systems.
  $\longrightarrow$ More than $7\times$ faster than GADGET-2 on a single core.
- Fully data-driven, asynchronous, and dynamic *hybrid shared/distributed-memory parallel model*.
  $\longrightarrow$ Good scaling even for small simulations on large numbers of cores.
- Domain decomposition based on *decomposing the task graph*.
  $\longrightarrow$ Decomposing the actual work, as opposed to just the data.

Conclusions

Our contributions

- Extended task-based parallelism conceptually to include *task conflicts*.
  $\longrightarrow$ Scheduling is trickier, but much more flexible.
- Faster and more cache-efficient algorithms for *neighbour-finding* in multi-scale particle systems.
  $\longrightarrow$ More than $7\times$ faster than GADGET-2 on a single core.
- Fully data-driven, asynchronous, and dynamic *hybrid shared/distributed-memory parallel model*.
  $\longrightarrow$ Good scaling even for small simulations on large numbers of cores.
- Domain decomposition based on *decomposing the task graph*.
  $\longrightarrow$ Decomposing the actual work, as opposed to just the data.

- Better algorithms/software leads to almost two orders of magnitude speedup.

Conclusions
Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.

Conclusions

Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.
  $\longrightarrow$ No need for exascale machines?

Conclusions

Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.
  $\longrightarrow$ No need for exascale machines?
  $\longrightarrow$ FLOPs are meaningless if they are being wasted on bad algorithms.

Conclusions

Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.
  $\longrightarrow$ No need for exascale machines?
  $\longrightarrow$ FLOPs are meaningless if they are being wasted on bad algorithms.
- Instead of telling Computer Scientists/Computer designers what they should be doing, maybe listen to what they have to say about computing.

Conclusions

Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.
  $\longrightarrow$ No need for exascale machines?
  $\longrightarrow$ FLOPs are meaningless if they are being wasted on bad algorithms.
- Instead of telling Computer Scientists/Computer designers what they should be doing, maybe listen to what they have to say about computing.

Conclusions

Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.
  $\longrightarrow$ No need for exascale machines?
  $\longrightarrow$ FLOPs are meaningless if they are being wasted on bad algorithms.
- Instead of telling Computer Scientists/Computer designers what they should be doing, maybe listen to what they have to say about computing.
- Paradigms and/or algorithms alone are a dime a dozen.

Conclusions

Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.
  $\longrightarrow$ No need for exascale machines?
  $\longrightarrow$ FLOPs are meaningless if they are being wasted on bad algorithms.
- Instead of telling Computer Scientists/Computer designers what they should be doing, maybe listen to what they have to say about computing.
- Paradigms and/or algorithms alone are a dime a dozen.
  $\longrightarrow$ Close collaborations are needed to produce useful software.

Conclusions

Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.
  $\longrightarrow$ No need for exascale machines?
  $\longrightarrow$ FLOPs are meaningless if they are being wasted on bad algorithms.
- Instead of telling Computer Scientists/Computer designers what they should be doing, maybe listen to what they have to say about computing.
- Paradigms and/or algorithms alone are a dime a dozen.
  $\longrightarrow$ Close collaborations are needed to produce useful software.
- Where do we go from here?

Conclusions

Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.
  $\longrightarrow$ No need for exascale machines?
  $\longrightarrow$ FLOPs are meaningless if they are being wasted on bad algorithms.
- Instead of telling Computer Scientists/Computer designers what they should be doing, maybe listen to what they have to say about computing.
- Paradigms and/or algorithms alone are a dime a dozen.
  $\longrightarrow$ Close collaborations are needed to produce useful software.
- Where do we go from here?
  $\longrightarrow$ More physics, better *generalized* vectorization, new architectures.

Conclusions

Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.
  $\longrightarrow$ No need for exascale machines?
  $\longrightarrow$ FLOPs are meaningless if they are being wasted on bad algorithms.
- Instead of telling Computer Scientists/Computer designers what they should be doing, maybe listen to what they have to say about computing.
- Paradigms and/or algorithms alone are a dime a dozen.
  $\longrightarrow$ Close collaborations are needed to produce useful software.
- Where do we go from here?
  $\longrightarrow$ More physics, better *generalized* vectorization, new architectures.

Conclusions

Take-home messages

- Better algorithms/software leads to almost two orders of magnitude speedup.
  $\longrightarrow$ No need for exascale machines?
  $\longrightarrow$ FLOPs are meaningless if they are being wasted on bad algorithms.
- Instead of telling Computer Scientists/Computer designers what they should be doing, maybe listen to what they have to say about computing.
- Paradigms and/or algorithms alone are a dime a dozen.
  $\longrightarrow$ Close collaborations are needed to produce useful software.
- Where do we go from here?
  $\longrightarrow$ More physics, better *generalized* vectorization, new architectures.
  $\longrightarrow$ Continue developing the task-based paradigm.

Thank you for your attention!