PALM - A Parallelized Large-Eddy Simulation Model for Atmospheric and Oceanic Flows on Knights Landing

Matthias Noack
noack@zib.de

Zuse Institute Berlin
Atmospheric Research with PALM

- Dustdevils
- Wind Energy
- Turbulence Effects on Aircraft
- Urban Meteorology and City Planning
- Cloud Physics

https://palm.muk.uni-hannover.de
The PALM Code

- continuously developed since 1997 by the PALM group (Siegfried Raasch et al.)
The PALM Code

- continuously developed since 1997 by the PALM group (Siegfried Raasch et al.)

- **Fortran** 95/2003.
- hybrid **MPI + OpenMP** code
- **140 kLOC**, 79 modules and 171 source files
- highly scalable, tested for up to 43,200 cores

https://palm.muk.uni-hannover.de
The PALM Code

- continuously developed since 1997 by the PALM group (Siegfried Raasch et al.)

- **Fortran** 95/2003.
- hybrid **MPI + OpenMP** code
- **140 kLOC**, 79 modules and 171 source files
- highly scalable, tested for up to 43,200 cores

- runs on the HLRN supercomputing facilities at Berlin (ZIB) and Hannover (LRZ)
- modernisation target within the **Intel Parallel Computing Center at ZIB**

https://palm.muk.uni-hannover.de
PALM Hackathon at ZIB

left to right: Matthias Noack, Florian Wende, Helge Knoop, Matthias Sühring, Tobias Gronemeier; behind the camera: Thomas Steinke
Parallelization Strategy

MPI

OpenMP
Parallelization Strategy

2D domain decomposition
Parallelization Strategy

2D domain decomposition

outer of 3 nested loops threaded

- different variants for different targets
- e.g. cache optimised with decomposed inner loop
- **no vectorisation**
  - use of *floating point exceptions* prevents automatic vectorisation
  - no explicit SIMD constructs

https://palm.muk.uni-hannover.de
KNL Optimisation Strategy

get operational on KNL
KNL Optimisation Strategy

- build scripts
- job scripts
- bug fixing
KNL Optimisation Strategy

- build scripts
- job scripts
- bug fixing
KNL Optimisation Strategy

- build scripts
- job scripts
- bug fixing

- small
- medium
- large
KNL Optimisation Strategy

- build scripts
- job scripts
- bug fixing

- define some benchmarks
  - small
  - medium
  - large

- measure baseline on Xeon
Benchmark Systems and Haswell Baseline

HLRN-III prod. system, 1872 nodes

- 2 × Intel Xeon E5-2680v3 (Haswell)
  - 2 × 12 cores at 2.5 GHz
  ⇒ **960 GFLOPS** per node
- 64 GiB DDR3, **136 GiB/s**
Benchmark Systems and Haswell Baseline

HLRN-III prod. system, 1872 nodes

- 2 × Intel Xeon E5-2680v3 (Haswell)
  - 2 × 12 cores at 2.5 GHz
  - \( \Rightarrow \) 960 GFLOPS per node
- 64 GiB DDR3, 136 GiB/s

Haswell Results

<table>
<thead>
<tr>
<th></th>
<th>nodes</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>4</td>
<td>131.3 s</td>
</tr>
<tr>
<td>medium</td>
<td>8</td>
<td>147.6 s</td>
</tr>
<tr>
<td>large</td>
<td>16</td>
<td>164.6 s</td>
</tr>
</tbody>
</table>
Benchmark Systems and Haswell Baseline

HLRN-III prod. system, 1872 nodes
- $2 \times$ Intel Xeon E5-2680v3 (Haswell)
  - $2 \times 12$ cores at 2.5 GHz
  - $960$ GFLOPS per node
- $64$ GiB DDR3, $136$ GiB/s

HLRN KNL TDS node, 80 nodes
- $1 \times$ Intel Xeon Phi 7250 (KNL)
  - 68 cores at 1.4 GHz
  - $2611.2$ GFLOPS with AVX clock "3.05 TFLOPS"
- $96$ GiB DDR4, $115.2$ GiB/s
- $16$ GiB MCDRAM, $490$ GiB/s

Haswell Results

<table>
<thead>
<tr>
<th></th>
<th>nodes</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>4</td>
<td>131.3 s</td>
</tr>
<tr>
<td>medium</td>
<td>8</td>
<td>147.6 s</td>
</tr>
<tr>
<td>large</td>
<td>16</td>
<td>164.6 s</td>
</tr>
</tbody>
</table>
Benchmark Systems and Haswell Baseline

**HLRN-III prod. system, 1872 nodes**

- 2 × Intel **Xeon** E5-2680v3 (Haswell)
  - 2 × 12 cores at 2.5 GHz
  - **960 GFLOPS** per node
- 64 GiB DDR3, **136 GiB/s**

**HLRN KNL TDS node, 80 nodes**

- 1 × Intel **Xeon Phi** 7250 (KNL)
  - 68 cores at 1.4 GHz
  - **2611.2 GFLOPS** with AVX clock "3.05 TFLOPS"
- 96 GiB DDR4, **115.2 GiB/s**
- 16 GiB **MCDRAM, 490 GiB/s**

**Haswell Results**

<table>
<thead>
<tr>
<th></th>
<th>nodes</th>
<th>runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>small</td>
<td>4</td>
<td>131.3 s</td>
</tr>
<tr>
<td>medium</td>
<td>8</td>
<td>147.6 s</td>
</tr>
<tr>
<td>large</td>
<td>16</td>
<td>164.6 s</td>
</tr>
</tbody>
</table>

- Upper bounds for speed-up:
  - compute bound: **2.7×**
  - memory bound: **3.6×** (MCDRAM)
KNL Optimisation Strategy

- build scripts
- job scripts
- bug fixing

- small
- medium
- large

✓
KNL Optimisation Strategy

- get operational on KNL
  - build scripts
  - job scripts
  - bug fixing

- define some benchmarks
  - small
  - medium
  - large

- measure baseline on Xeon

✓ MCDRAM and boot mode
MCDRAM Usage and Boot Mode

boot flat mode

run in DDR

run in MCDRAM

- problem < 16 GiB
- upper bound for gain from MCDRAM

boot cache mode

run again

⇒ decide about explicit placement

- good enough?
MCDRAM Usage and Boot Mode

- boot flat mode
- run in DDR
- run in MCDRAM
- boot cache mode
- run again

- problem < 16 GiB
- upper bound for gain from MCDRAM
- good enough? ⇒ decide about explicit placement

- 25 - 41% gain from MCDRAM
- ≤ 3% loss from Cache-Mode
  ⇒ no need for explicit placement
- 2 MiB pages worked best
**KNL Optimisation Strategy**

- **get operational on KNL**
  - build scripts
  - job scripts
  - bug fixing

- **define some benchmarks**
  - small
  - medium
  - large

- **measure baseline on Xeon**

  ✓

  ⇒ **MCDRAM and boot mode**

  ⇒ quad_cache
KNL Optimisation Strategy

get operational on KNL
- build scripts
- job scripts
- bug fixing

define some benchmarks
- small
- medium
- large

measure baseline on Xeon
✓
⇒ quad_cache

MCDRAM and boot mode

MPI processes vs. OpenMP threads
### MPI processes vs. OpenMP threads

#### Tuning run for optimal per-node config

<table>
<thead>
<tr>
<th>ranks</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

- **small**
  - ≤ 2% off from best
  - X: worst

- **medium**
  - ≤ 15% off from best
  - X: fastest

- **large**
  - ≤ 2% off from best
  - X: worst
### MPI processes vs. OpenMP threads

#### Tuning run for optimal per-node config

<table>
<thead>
<tr>
<th>ranks</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>small</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>large</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

- **X**: fastest
- ≤ 2% off from best
- ≤ 15% off from best
- worse

#### Conclusion

- **fastest**
  - small: 16 ranks × 4 thread
  - medium: 8 ranks × 8 threads
  - large: 32 × 2 threads
- 16 × 4 performs for all setups
- fastest vs. slowest config: **2.3×**

quad_cache mode
MPI processes vs. OpenMP threads

Tuning run for optimal per-node config

<table>
<thead>
<tr>
<th>ranks</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>small</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>medium</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>large</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- ≤ 2% off from best
- ≤ 15% off from best
- worst
- fastest

Conclusion

- fastest
  - small: 16 ranks × 4 thread
  - medium: 8 ranks × 8 threads
  - large: 32 × 2 threads
- 16 × 4 performs for all setups
- fastest vs. slowest config: 2.3×
- very low impact on speedups between MCDRAM usage models
  - absolute numbers vary largely
  - do memory first
### Tuning run for optimal per-node config

<table>
<thead>
<tr>
<th>ranks</th>
<th>64</th>
<th>32</th>
<th>16</th>
<th>8</th>
<th>4</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>small</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>medium</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>large</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **green**: ≤ 2% off from best
- **orange**: ≤ 15% off from best
- **red**: worse
- **red X**: fastest

### Conclusion

- **fastest**
  - small: 16 ranks × 4 thread
  - medium: 8 ranks × 8 threads
  - large: 32 × 2 threads
- 16 × 4 performs for all setups
- fastest vs. slowest config: **2.3×**
- very low impact on speedups between MCDRAM usage models
  - absolute numbers vary largely
  - do memory first
- always check pinning/affinity
KNL Optimisation Strategy

- get operational on KNL
  - build scripts
  - job scripts
  - bug fixing

- define some benchmarks
  - small
  - medium
  - large

- measure baseline on Xeon
  ✓

- MCDRAM and boot mode
  ⇒ quad_cache

... ⇒ 16 × 4

MPI processes vs. OpenMP threads

...
KNL Optimisation Strategy

- build scripts
- job scripts
- bug fixing

- define some benchmarks
  - small
  - medium
  - large

- measure baseline on Xeon

- MCDRAM and boot mode

⇒ quad_cache

MPI processes vs. OpenMP threads

⇒ 16 × 4

start optimisation workflow
KNL Optimisation Strategy

get operational on KNL
- build scripts
- job scripts
- bug fixing

define some benchmarks
- small
- medium
- large

measure baseline on Xeon
✓ ⇒ quad_cache

MCDRAM and boot mode

MPI processes vs. OpenMP threads
⇒ 16 × 4

start optimisation workflow

... ⇒ ...
**KNL Optimisation Strategy**

- get operational on KNL
  - build scripts
  - job scripts
  - bug fixing

- define some benchmarks
  - small
  - medium
  - large

- measure baseline on Xeon

- MCDRAM and boot mode

- quad_cache

- MPI processes vs. OpenMP threads
  ⇒ 16 × 4

- compiler optimisation reports
- Intel VTune Amplifier XE, Advisor XE
First Code Changes

Floating point exception-handling

- prevents vectorisation
- `fp-model-strict → fp-model-source`
- remove exception handling
- add NaN/Inf-tests
  - when writing checkpoints
- no significant auto vectorisation
- suboptimal memory layout
First Code Changes

Floating point exception-handling

- prevents vectorisation
- \texttt{fp-model-strict} $\rightarrow$ \texttt{fp-model-source}
- remove exception handling
- add NaN/Inf-tests
  - $\Rightarrow$ when writing checkpoints
  - $\Rightarrow$ no significant auto vectorisation
  - $\Rightarrow$ suboptimal memory layout

Using MKL FFT

- small gain for benchmarks
  - $\Rightarrow$ might be significant for larger setups
First Code Changes

Floating point exception-handling

- prevents vectorisation
- **fp-model-strict** → **fp-model-source**
- remove exception handling
- add NaN/Inf-tests
  ⇒ when writing checkpoints
⇒ no significant auto vectorisation
⇒ suboptimal memory layout

Using MKL FFT

- small gain for benchmarks
⇒ might be significant for larger setups

CONTIGUOUS keyword

- from Fortran 2008
- tell the compiler about contiguously allocated arrays

Current Results, Intel Compiler 17.0.0

<table>
<thead>
<tr>
<th></th>
<th>small</th>
<th>medium</th>
<th>large</th>
</tr>
</thead>
<tbody>
<tr>
<td>HSW, Baseline</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>HSW, Current</td>
<td>1.09</td>
<td>1.13</td>
<td>1.01</td>
</tr>
<tr>
<td>KNL, Current</td>
<td>1.24</td>
<td>1.07</td>
<td>0.96</td>
</tr>
</tbody>
</table>
Cray Specialities

Hardware/Software

- Cray Aries network
- Cray MPI
- Cray Performance Tools
- Cray Compiler
Cray Specialities

Hardware/Software

- Cray Aries network
- Cray MPI
- Cray Performance Tools
- Cray Compiler

Rank Reordering

- instrumented application run
- optimised mapping of MPI ranks to cores and nodes

⇒ no improvement on KNL
Cray Specialities

Hardware/Software

- Cray Aries network
- Cray MPI
- Cray Performance Tools
- Cray Compiler

Rank Reordering

- instrumented application run
- optimised mapping of MPI ranks to cores and nodes
  \[\Rightarrow\text{no improvement on KNL}\]

Cray Compiler

- initial KNL support
- crash with OpenMP
  \[\Rightarrow64\text{ MPI ranks per KNL}\]
Projected Production Run Performance

- benchmark runs: ≈ 5 min, productions runs: ≈ 12 hours
  - serial initialisation becomes negligible
  - plot speedup based on $t_{total} - t_{init}$
Projected Production Run Performance

- benchmark runs: $\approx 5 \text{ min}$, productions runs: $\approx 12 \text{ hours}$
  - serial initialisation becomes negligible
  - plot speedup based on $t_{\text{total}} - t_{\text{init}}$
Final Remarks

Bottom Line

- getting started on KNL was easy
  ⇒ way easier than KNC and offloading
- good initial performance (cache-mode)
- scalar parts hurt
  ⇒ initialisation
  ⇒ ...
Final Remarks

Bottom Line

- getting started on KNL was easy
  ⇒ way easier than KNC and offloading
- good initial performance (cache-mode)
- scalar parts hurt
  ⇒ initialisation
  ⇒ ...
- Cray Fortran compiler up to 16.5 %
  faster than Intel on KNL

https://palm.muk.uni-hannover.de
Final Remarks

Bottom Line

- getting started on KNL was easy
  ⇒ way easier than KNC and offloading
- good initial performance (cache-mode)
- scalar parts hurt
  ⇒ initialisation
  ⇒ ...
- Cray Fortran compiler up to 16.5% faster than Intel on KNL
- speedup over dual HSW HLRN node:
  - benchmark: up to 1.29×
  - production (projected): up to 1.45×
  ⇒ even without vectorisation

https://palm.muk.uni-hannover.de
Final Remarks

Bottom Line

- getting started on KNL was easy ⇒ way easier than KNC and offloading
- good initial performance (cache-mode)
- scalar parts hurt ⇒ initialisation ⇒ ...
- Cray Fortran compiler up to 16.5% faster than Intel on KNL
- speedup over dual HSW HLRN node:
  - benchmark: up to 1.29×
  - production (projected): up to 1.45× ⇒ even without vectorisation

What’s next

- VTune Results:
  ⇒ increase concurrency
  ⇒ reduce L2 misses on KNL
- adapt data layout for SIMD
  ⇒ twice the potential on KNL
Thank you.