Enhancing task-based irregular OpenMP applications with the libKOMP runtime system

François Broquedis
MOAIS Team, Grenoble, France
Working in the MOAIS team

• Many research directions
  ‣ Parallel algorithms / scheduling
  ‣ Interactive computing and visualization
  ‣ Runtime systems for HPC platforms

• I’m mainly involved in the third one
  ‣ Task-based parallelism (kaapi runtime system)
  ‣ OpenMP as a mean to express task-based parallelism (libKOMP runtime system)
OUTLINE

1. A brief introduction to OpenMP task parallelism
2. Implementing the OpenMP tasking model
3. Adaptive loop scheduling for OpenMP
4. Data affinity and task-based applications
5. Current status from the HPC-GA project perspective
OUTLINE

1. A brief introduction to OpenMP task parallelism
2. Implementing the OpenMP tasking model
3. Adaptive loop scheduling for OpenMP
4. Data affinity and task-based applications
5. Current status from the HPC-GA project perspective
OUTLINE

A longer introduction to OpenMP (in general)

1. A brief introduction to OpenMP task parallelism
2. Implementing the OpenMP tasking model
3. Adaptive loop scheduling for OpenMP
4. Data affinity and task-based applications
5. Current status from the HPC-GA project perspective
What Is OpenMP?

- *De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran*

- *Consists of Compiler Directives, Run time routines and Environment variables*

- *Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)*

- *Version 3.1 has been released July 2011*
  - *The upcoming 4.0 specifications will be released soon*
OpenMP is widely supported by the industry, as well as the academic community.
Advantages Of OpenMP

- Good performance and scalability
  - If you do it right ....
- De-facto and mature standard
- An OpenMP program is portable
  - Supported by a large number of compilers
- Requires little programming effort
- Allows the program to be parallelized incrementally
The OpenMP Execution Model

Fork and Join Model

Master Thread

Parallel region

Worker Threads

Synchronization

Parallel region

Worker Threads

Synchronization
Nested Parallelism

Master Thread

3-way parallel

9-way parallel

3-way parallel

Note: Nesting level can be arbitrarily deep

Outer parallel region

Nested parallel region

Outer parallel region

vendredi 15 mars 13
The OpenMP Memory Model

- All threads have access to the same, globally shared memory
- Data in private memory is only accessible by the thread owning this memory
- No other thread sees the change(s)
- Data transfer is through shared memory and is 100% transparent to the application
Data-Sharing Attributes

- In an OpenMP program, data needs to be "labeled"
- Essentially there are two basic types:
  - **Shared** - There is only one instance of the data
    - Threads can read and write the data simultaneously unless protected through a specific construct
    - All changes made are visible to all threads
      - But not necessarily immediately, unless enforced ......
  - **Private** - Each thread has a copy of the data
    - No other thread can access this data
    - Changes only visible to the thread owning the data
The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless nowait is specified)
- A work-sharing construct does not launch any new threads
The Schedule Clause

**schedule ( static | dynamic | guided | auto [, chunk] )**

**schedule ( runtime )**

**static [, chunk]**

- **Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion**

- **In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads**
  - Details are implementation defined

- **Under certain conditions, the assignment of iterations to threads is the same across multiple loops in the same parallel region**
Example Of A Static Schedule

A loop of length 16 using 4 threads

<table>
<thead>
<tr>
<th>Thread</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>no chunk *</td>
<td>1-4</td>
<td>5-8</td>
<td>9-12</td>
<td>13-16</td>
</tr>
<tr>
<td>chunk = 2</td>
<td>1-2</td>
<td>3-4</td>
<td>5-6</td>
<td>7-8</td>
</tr>
<tr>
<td></td>
<td>9-10</td>
<td>11-12</td>
<td>13-14</td>
<td>15-16</td>
</tr>
</tbody>
</table>

*) The precise distribution is implementation defined
Loop Workload Scheduling Choices

**dynamic [, chunk]**
- Fixed portions of work; size is controlled by the value of chunk
- When a thread finishes, it starts on the next portion of work

**guided [, chunk]**
- Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially

**auto**
- The compiler (or runtime system) decides what is best to use; choice could be implementation dependent

**runtime**
- Iteration scheduling scheme is set at runtime through environment variable `OMP_SCHEDULE`
Experiment – 500 Iterations, 4 Threads

- Guided: 5 threads
- Dynamic: 5 iterations
- Static
A gentle introduction to parallel programming and HPC platforms

A first example

\[ f = 1.0 \]

\[
\text{for } (i = 0; i < n; i++) \\
\quad z[i] = x[i] + y[i];
\]

\[
\text{for } (i = 0; i < m; i++) \\
\quad a[i] = b[i] + c[i];
\]

\[ \ldots \]

\[ \text{scale} = \text{sum} \ (a, 0, m) + \text{sum} \ (z, 0, n) + f; \]
\[ \ldots \]
A first example - v1

```c
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
    f = 1.0

    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];

    for (i = 0; i < m; i++)
        a[i] = b[i] + c[i];

    ...

    scale = sum (a, 0, m) + sum (z, 0, n) + f;
    ...
} /* End of OpenMP parallel region */
```
A Gentle Introduction to Parallel Programming and HPC Platforms

A first example - v1

```c
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
    f = 1.0
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];
    for (i = 0; i < m; i++)
        a[i] = b[i] + c[i];
    ...
    scale = sum (a, 0, m) + sum (z, 0, n) + f;
    ...
} /* End of OpenMP parallel region */
```

Statements executed by all the threads of the parallel region!
A gentle introduction to parallel programming and HPC platforms

A first example - v1

```c
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale)
{
    f = 1.0

#pragma omp for
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];

#pragma omp for
    for (i = 0; i < m; i++)
        a[i] = b[i] + c[i];

    scale = sum (a, 0, m) + sum (z, 0, n) + f;

} /* End of OpenMP parallel region */
```
A gentle introduction to parallel programming and HPC platforms

A first example - v2

```c
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m) private (f, i, scale)
{
    f = 1.0

    #pragma omp for nowait
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];

    #pragma omp for nowait
    for (i = 0; i < m; i++)
        a[i] = b[i] + c[i];

    ...

    #pragma omp barrier
    scale = sum (a, 0, m) + sum (z, 0, n) + f;
    ...

} /* End of OpenMP parallel region */
```
A first example - v3

```c
#pragma omp parallel default (none) shared (z, x, y, a, b, c, n, m)
private (f, i, scale) if (n > some_threshold && m > some_threshold) {
    f = 1.0

#pragma omp for nowait
    for (i = 0; i < n; i++)
        z[i] = x[i] + y[i];

#pragma omp for nowait
    for (i = 0; i < m; i++)
        a[i] = b[i] + c[i];

...  

#pragma omp barrier
    scale = sum (a, 0, m) + sum (z, 0, n) + f;
    ...
} /* End of OpenMP parallel region */
```
More processing units means massive parallelism

- Smaller chip designs mean bigger HPC platforms...
  - More and more cores per chip
  - More and more chips per nodes

- ... that can turn out to be embarrassing!
  - How can I occupy all these cores?

- The programmer has to express fine-grain massive parallelism
  - OpenMP has evolved to fill this need with the 3.0 standard introducing OpenMP tasks (2008)
The tasking concept in OpenMP
OpenMP tasking: Who does what and when? (Don’t ask why...)

• The developer specifies where the tasks are
  ‣ #pragma omp task construct

• The assumption is that all tasks can be executed independently

• When any thread encounters a task construct, an explicit task is generated
  ‣ Tasks can be nested

• Execution of explicitly generated tasks is assigned to one of the threads of the current team
  ‣ This is subject to the thread’s availability and thus could be immediate or deferred until later

• Completion of the task can be guaranteed using the taskwait synchronization construct
The Fibonacci example

```c
int main (int argc, char **argv)
{
  long long par_res = 0;

#pragma omp parallel
{
  #pragma omp single
  par_res = fib(n);
}

  printf ("Fibonacci result for %i is %lli\n", n, par_res);
  return EXIT_SUCCESS;
}
```

```c
long long fib (int n)
{
  long long x, y;
  if (n < 2)
    return n;

#pragma omp task shared(x)
  firstprivate(n)
    x = fib(n - 1);

#pragma omp task shared(y)
  firstprivate(n)
    y = fib(n - 2);

#pragma omp taskwait
    return x + y;
}
```
OUTLINE

1. A brief introduction to OpenMP task parallelism
2. Implementing the OpenMP tasking model
3. Adaptive loop scheduling for OpenMP
4. Data affinity and task-based applications
5. Current status from the HPC-GA project perspective
The work-stealing execution model

• Work-stealing = efficient way of executing task-based parallelism
  ‣ Dynamic load balancing triggered by idle processing units
    ○ Per-core queues of tasks
    ○ Idle cores «steal» ready tasks from occupied queues
  ‣ Scheduling decisions
    ○ Next task to execute?
    ○ Core to steal from?

• Programming environments based on work-stealing
  ‣ Cilk, TBB
The way libKOMP executes tasks

• One «worker thread» per core
  ‣ Able to execute libKOMP fine-grain tasks
  ‣ Holds a queue of tasks
    ◦ Related to sequential C stack of activation frames
    ◦ Last-in first-out data structure

• Task creation is cheap!
  ‣ Reduces to pushing C function pointer + its arguments into the worker thread queue
  ‣ Recursive tasks are welcome!

• Work-stealing based scheduling
  ‣ Cilks’s work first principle
  ‣ Work-stealing algorithm = plug-in
    ◦ Default: steal a task from a randomly chosen queue
libKOMP: an OpenMP runtime system based on kaapi

- **OpenMP applications generate kaapi tasks**
  - OpenMP tasks == kaapi tasks
  - Parallel region create num_threads kaapi tasks
    - Nested parallel regions create recursive tasks
  - Worker threads created at library initialization

- **The libKOMP library**
  - Port of the libGOMP ABI on top of the kaapi runtime system
  - Binary compatible with existing gcc-compiled OpenMP application
# Performance evaluation with BOTS

## Barcelona OpenMP Task Suite

- A set of representative benchmarks to evaluate OpenMP tasks implementations

<table>
<thead>
<tr>
<th>Name</th>
<th>Arguments used</th>
<th>Domain</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>prot100.aa</td>
<td>Dynamic programming</td>
<td>Aligns sequences of proteins</td>
</tr>
<tr>
<td>FFT</td>
<td>n=33,554,432</td>
<td>Spectral method</td>
<td>Computes a Fast Fourier Transformation</td>
</tr>
<tr>
<td>Floorplan</td>
<td>input.20</td>
<td>Optimization</td>
<td>Computes the optimal placement of cells in a floorplan</td>
</tr>
<tr>
<td>NQueens</td>
<td>n=14</td>
<td>Search</td>
<td>Finds solutions of the N Queens problem</td>
</tr>
<tr>
<td>MultiSort</td>
<td>n=33,554,432</td>
<td>Integer sorting</td>
<td>Uses a mixture of sorting algorithms to sort a vector</td>
</tr>
<tr>
<td>SparseLU</td>
<td>n=128 m=64</td>
<td>Sparse linear algebra</td>
<td>Computes the LU factorization of a sparse matrix</td>
</tr>
<tr>
<td>Strassen</td>
<td>n=8192</td>
<td>Dense linear algebra</td>
<td>Computes a matrix multiply with Strassen's method</td>
</tr>
<tr>
<td>UTS</td>
<td>medium.input</td>
<td>Search</td>
<td>Computes the number of nodes in an Unbalanced Tree</td>
</tr>
</tbody>
</table>

## Evaluation platform

- AMD48: 4x12 AMD Opteron cores

## Softwares

- gcc 4.6.2 + libGOMP
- gcc 4.6.2 + libKOMP
- icc 12.1.2 + Intel OpenMP runtime (KMP)
Running BOTS on the AMD48 platform

<table>
<thead>
<tr>
<th>kernel</th>
<th>libGOMP</th>
<th>libKOMP</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>38.8</td>
<td>40.0</td>
<td>37.0</td>
</tr>
<tr>
<td>FFT</td>
<td>0.5</td>
<td>12.2</td>
<td>12.0</td>
</tr>
<tr>
<td>Floorplan</td>
<td>27.6</td>
<td>32.7</td>
<td>29.2</td>
</tr>
<tr>
<td>NQueens</td>
<td>43.7</td>
<td>43.4</td>
<td>39.0</td>
</tr>
<tr>
<td>MultiSort</td>
<td>0.6</td>
<td>13.2</td>
<td>11.3</td>
</tr>
<tr>
<td>SparseLU</td>
<td>44.1</td>
<td>44.4</td>
<td>35.0</td>
</tr>
<tr>
<td>Strassen</td>
<td>20.8</td>
<td>22.4</td>
<td>20.5</td>
</tr>
<tr>
<td>UTS</td>
<td>0.9</td>
<td>25.3</td>
<td>15.0</td>
</tr>
</tbody>
</table>

Speed-Up of BOTS kernels on the 48 cores of the AMD48 platform
Running BOTS on the AMD48 platform

<table>
<thead>
<tr>
<th>kernel</th>
<th>libGOMP</th>
<th>libKOMP</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>38.8</td>
<td>40.0</td>
<td>37.0</td>
</tr>
<tr>
<td>FFT</td>
<td>0.5</td>
<td>12.2</td>
<td>12.0</td>
</tr>
<tr>
<td>Floorplan</td>
<td>27.6</td>
<td>32.7</td>
<td>29.2</td>
</tr>
<tr>
<td>NQueens</td>
<td>43.7</td>
<td>43.4</td>
<td>39.0</td>
</tr>
<tr>
<td>MultiSort</td>
<td>0.6</td>
<td>13.2</td>
<td>11.3</td>
</tr>
<tr>
<td>SparseLU</td>
<td>44.1</td>
<td>44.4</td>
<td>35.0</td>
</tr>
<tr>
<td>Strassen</td>
<td>20.8</td>
<td>22.4</td>
<td>20.5</td>
</tr>
<tr>
<td>UTS</td>
<td>0.9</td>
<td>25.3</td>
<td>15.0</td>
</tr>
</tbody>
</table>

Speed-Up of BOTS kernels on the 48 cores of the AMD48 platform
OUTLINE

1. A brief introduction on OpenMP task parallelism
2. Implementing the OpenMP tasking model
3. Adaptive loop scheduling for OpenMP
4. Data affinity and task-based applications
5. Current status from the HPC-GA project perspective
The problem of granularity

- OpenMP tasks = a good candidate to efficiently exploit multicore chips
  - More and more cores per chip
  - Fine-grain parallelism
    - Tasks are lighter than threads

- Problem: creating the «right» number of tasks for a specific platform
  - Too many tasks => scheduling overheads
  - Not enough tasks => load imbalance
  - Old problem for the OpenMP community!
    - num_threads and nested parallel regions
Adaptive OpenMP loop scheduling with libKOMP

• Adaptive tasks in XKaapi
  ‣ Adaptative tasks can be split at run time to create new tasks
  ‣ Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

• Exemple of use: the way libKOMP executes OpenMP parallel loops
  ‣ Adaptive tasks for independent loops
    ◦ A task == a range of iterations to compute
  ‣ Execution model
    ◦ Initially, one task in charge of the whole range

\[ T_1 : [0 - 15] \]
Adaptive OpenMP loop scheduling with libKOMP

- Adaptive tasks in XKaapi
  - Adaptative tasks can be split at run time to create new tasks
  - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

- Exemple of use: the way libKOMP executes OpenMP parallel loops
  - Adaptive tasks for independent loops
    - A task == a range of iterations to compute
  - Execution model
    - Initially, one task in charge of the whole range

\[ T_1 : [1 - 15] \]
Adaptive OpenMP loop scheduling with libKOMP

• Adaptive tasks in XKaapi
  ‣ Adaptative tasks can be split at run time to create new tasks
  ‣ Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

• Exemple of use: the way libKOMP executes OpenMP parallel loops
  ‣ Adaptive tasks for independent loops
    ☐ A task == a range of iterations to compute
  ‣ Execution model
    ☐ Initially, one task in charge of the whole range

\[ T_1 : [2 \text{ - } 15] \]
Adaptive OpenMP loop scheduling with libKOMP

- Adaptive tasks in XKaapi
  - Adaptative tasks can be split at run time to create new tasks
  - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

- Exemple of use: the way libKOMP executes OpenMP parallel loops
  - Adaptive tasks for independent loops
    - A task == a range of iterations to compute
  - Execution model
    - Initially, one task in charge of the whole range

\[ T_1 : [3 - 15] \]
Adaptive OpenMP loop scheduling with libKOMP

• Adaptive tasks in XKaapi
  ▸ Adaptative tasks can be split at run time to create new tasks
  ▸ Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

• Exemple of use: the way libKOMP executes OpenMP parallel loops
  ▸ Adaptive tasks for independent loops
    ☐ A task == a range of iterations to compute
  ▸ Execution model
    ☐ Initially, one task in charge of the whole range
    ☐ Idle cores post steal requests and trigger the “split” operation to generate new tasks

Idle Core

steal request

split (T₁)

T₁ : [4 - 15]
Adaptive OpenMP loop scheduling with libKOMP

- Adaptive tasks in XKaapi
  - Adaptative tasks can be split at run time to create new tasks
  - Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

- Example of use: the way libKOMP executes OpenMP parallel loops
  - Adaptive tasks for independent loops
    - A task == a range of iterations to compute
  - Execution model
    - Initially, one task in charge of the whole range
    - Idle cores post steal requests and trigger the “split” operation to generate new tasks

Idle Core

steal request

split (T₁)

T₁ : [4 - 15]
Adaptive OpenMP loop scheduling with libKOMP

• Adaptive tasks in XKaapi
  ‣ Adaptative tasks can be split at run time to create new tasks
  ‣ Provide a “splitter” function called when an idle core decides to steal some of the remaining computation to be performed by a task under execution

• Example of use: the way libKOMP executes OpenMP parallel loops
  ‣ Adaptive tasks for independent loops
    ☰ A task == a range of iterations to compute
  ‣ Execution model
    ☰ Initially, one task in charge of the whole range
    ☰ Idle cores post steal requests and trigger the “split” operation to generate new tasks

Idle Core
steal request

split (T₁)

T₁ : [4 - 9]  T₂ : [10 - 15]
Performance evaluation: adaptive loops to enhance VTK filters [Mathias Ettinger, PhD student]

- Parallel version of the VTK visualization toolkit
  - A framework to develop parallel applications for scientific visualization
  - A pipeline of VTK «filters»
    - A filter == a computation performed on a 2D/3D scene

- Parallelized filters
  - Iso-Surface
    - for (all points p in my scene)
      - if (p meets some requirements)
        - compute_smthg (p);
    - irregular workload
Performance evaluation: adaptive loops to enhance VTK filters [Mathias Ettinger, PhD student]

Performance of the VTKcontour filter on a 48-core AMD shared memory machine

- libKOMP (adaptive)
- OpenMP (static)
- OpenMP (dynamic, 32)
- OpenMP (dynamic, 512)
- OpenMP (dynamic, 1024)
- OpenMP (dynamic, 5300)
OUTLINE

1. A brief introduction on OpenMP task parallelism
2. Implementing the OpenMP tasking model
3. Adaptive loop scheduling for OpenMP
4. Data affinity and task-based applications
5. Current status from the HPC-GA project perspective
The NUMA problem

- Non-Uniform Memory Accesses
  - The NUMA factor
  - Write access = more traffic

- Memory affinity
  - Applications run faster if accessing local data
  - Need a careful distribution of threads and data
    - To avoid NUMA penalties
    - To reduce memory contention

<table>
<thead>
<tr>
<th>Access to...</th>
<th>Local node</th>
<th>Neighbor node</th>
<th>Opposite node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>83 ns</td>
<td>98 ns (x1.18)</td>
<td>117 ns (x1.41)</td>
</tr>
<tr>
<td>Write</td>
<td>142 ns</td>
<td>177 ns (x1.25)</td>
<td>208 ns (x1.46)</td>
</tr>
</tbody>
</table>
Different machines, different NUMA topologies

Results of the STREAM benchmark (memory bandwidth in MB/s) for two different shared-memory machines

(a) 4 AMD 6174 sockets – 2 nodes per socket – 6 cores per node (48 cores total)

(b) 4 Intel X7560 – 8 cores per node
Parallel programming and memory-bound problems

- Usually a matter of...
  - ... binding memory on specific NUMA nodes ...
  - ... and binding threads according to the memory mapping
    - on the same NUMA node the data they access were allocated on

- What about task-based irregular applications?
  - Tasks mapping will change dynamically
    - Tasks are meant to move to balance the application workload
  - Most common solution: the *interleave* policy
    - Distribute the memory pages of the data set over the whole machine to minimize the probability of memory contention
Controlling task execution on hierarchical architectures

- Let the programmer...
  - Bind memory on specific nodes
  - Settle tasks on «locality domains»
    - Tasks settled on a locality domain can only be stolen by cores from the same locality domain
    - If the stealing process fails, the idle core extends its research scope to a more general (upper-level) locality domain
    - Locality domains are inherited by recursive children tasks

Default locality domain (the whole machine)
Controlling task execution on hierarchical architectures

- Let the programmer...
  - Bind memory on specific nodes
  - Settle tasks on «locality domains»
    - Tasks settled on a locality domain can only be stolen by cores from the same locality domain
    - If the stealing process fails, the idle core extends its research scope to a more general (upper-level) locality domain
    - Locality domains are inherited by recursive children tasks

- Core
  - L3
  - NUMA bank

- Core
  - L3
  - NUMA bank

« → » means «creates» or «is the father of»
Controlling task execution on hierarchical architectures

- Let the programmer...
  - Bind memory on specific nodes
  - Settle tasks on «locality domains»
    - Tasks settled on a locality domain can only be stolen by cores from the same locality domain
    - If the stealing process fails, the idle core extends its research scope to a more general (upper-level) locality domain
    - Locality domains are inherited by recursive children tasks

```c
kaapi_attr_set_locality_domain (&attr, NUMA_NODE_0);
```

**Default locality domain (the whole machine)**

**User-specified locality domain (NUMA node 0)**

```
<table>
<thead>
<tr>
<th>Core</th>
<th>Core</th>
<th>Core</th>
<th>Core</th>
<th>Core</th>
<th>Core</th>
<th>Core</th>
<th>Core</th>
</tr>
</thead>
<tbody>
<tr>
<td>L3</td>
<td>L3</td>
<td>L3</td>
<td>L3</td>
<td>L3</td>
<td>L3</td>
<td>L3</td>
<td>L3</td>
</tr>
<tr>
<td>NUMA bank</td>
<td>NUMA bank</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Controlling task execution on hierarchical architectures

- Let the programmer...
  - Bind memory on specific nodes
  - Settle tasks on «locality domains»
    - Tasks settled on a locality domain can only be stolen by cores from the same locality domain
    - If the stealing process fails, the idle core extends its research scope to a more general (upper-level) locality domain
    - Locality domains are inherited by recursive children tasks

```c
kaapi_attr_set_locality_domain (&attr, NUMA_NODE_0);
```

![Diagram showing task execution and locality domains](attachment:image.png)
Applying this approach to PMA sorting [Marie Durand, PhD student]

- Efficient data structure for sorting particles
  - Packed Memory Array (PMA):
    Add «gaps» to the particle array to minimize memory movement induced by reordering

- On-going work: Parallel PMA
  - Partition the PMA data structure to apply reordering in parallel
    - recursively create tasks that will scan and sort a subpart of the partition
Early results with task/data affinities

Fig. 4. Speedup of sort operation on the Intel-32 machine.
OUTLINE

1. A brief introduction on OpenMP task parallelism
2. Implementing the OpenMP tasking model
3. Adaptive loop scheduling for OpenMP
4. Data affinity and task-based applications
5. Current status from the HPC-GA project perspective
Summary - The libKOMP runtime system

• Efficient implementation of OpenMP independant tasks...
  ‣ Low runtime-related overheads
    ☐ Cheap task creation
  ‣ Provide support for recursive tasks
  ‣ Performance comparable to «mainstream» OpenMP runtime systems
  ‣ Easy to use
    ☐ Binary compatible with gcc-compiled OpenMP applications

• ... that comes with original features
  ‣ Adaptive loop scheduling
  ‣ Locality domains for hierarchical architectures

• We started collaborations with people from the HPC-GA project
  ‣ Parallelization of the Hou10ni application (Julien Diaz, Giannis Ashiotis)
  ‣ Early experiments with the Ondes3D application (Fabrice Dupros)
  ‣ No physicists were harmed during these experiments!
Newcomers are welcome!

• Remember libKOMP is a (young) research project
  ‣ Lots of bugs fixed thanks to the Hou10ni + Ondes3D experience
    ○ A new software distribution is on its way included all these fixes
  
  ‣ The «task part» of OpenMP is usually easier to handle for us than the «thread part»
    ○ The underlying task-based runtime system was not designed to mimic threads

• The approach we recommend
  ‣ Develop and tune your application with standard OpenMP
    ○ We can give you some advice if you’re not familiar with it
  ‣ In a second step, experiment with libKOMP’s original features
    ○ If it does not behave as expected, you at least get an efficient standard OpenMP version of your app! :-)

Thank you for your attention!
Applying this approach to PMA sorting [Marie Durand, PhD student]

- Studying particle interactions for physics simulations
  - Short-range interactions between particles
  - Only some of them will move
  - Many interactive-computing applications iterate over all the particles of a 3D domain
Applying this approach to PMA sorting [Marie Durand, PhD student]

- Studying particle interactions for physics simulations
  - Short-range interactions between particles
  - Only some of them will move
  - Many interactive-computing applications iterate over all the particles of a 3D domain

Group particles by cells
Applying this approach to PMA sorting [Marie Durand, PhD student]

• Studying particle interactions for physics simulations
  › Short-range interactions between particles
  › Only some of them will move
  › Many interactive-computing applications iterate over all the particles of a 3D domain

• Cache-oblivious data structures for moving particles
  › Group particles by cells
  › Keep this set of cells sorted
    ○ Z-curve ordering, maximize locality
Applying this approach to PMA sorting [Marie Durand, PhD student]

• Efficient data structure for sorting particules
  ⚫ Packed Memory Array (PMA):
    Add «gaps» to the particle array to minimize memory movement induced by reordering

Move 5 elements
Move 2 elements
Applying this approach to PMA sorting [Marie Durand, PhD student]

- Efficient data structure for sorting particules
  - Packed Memory Array (PMA):
    Add «gaps» to the particle array to minimize memory movement induced by reordering

- On-going work: Parallel PMA [Marie Durand, PhD student]
  - Partition the PMA data structure to apply reordering in parallel
    - recursively create tasks that will scan and sort a subpart of the partition
OpenMP tasks and synchronizations

- OpenMP let the application programmer in charge of tasks synchronization
  - #pragma omp taskwait construct
  - Wait of the completion of child tasks of the current task
    - Here, wait for A and B before executing C and D

- How to express the fact that task C can start executing as soon as task A is complete?

```c
void data_flow_example (void)
{
    type x, y;

    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task
        compute_smthg (&x);  /* Task A */
        #pragma omp task
        compute_smthg (&y);  /* Task B */
    }
    #pragma omp taskwait

    #pragma omp task
    process_results (x);  /* Task C */
    #pragma omp task
    process_results (y);  /* Task D */
}
```
Data-flow parallelism with libKOMP

• Data-flow execution model
  ▸ The application programmer expresses dependencies between tasks
    ◊ Declare variables / memory areas the task will use as input and/or produce as output
      - Access modes for variables (read, write)

  ▸ The runtime system computes a data-flow graph to decide whether a task is ready to be executed or not
    ◊ Synchronization is no longer the application programmer’s burden!

• libKOMP’s execution model for data-flow parallelism
  ▸ A task is ready for execution when all its input variables are ready
  ▸ A variable is ready as soon as it’s been written
libKOMP’s API for data-flow programming

• **Keywords defining variable access modes**
  - read: variable is read as input
  - write: variable is written as output
  - readwrite: variable is both read as input and written as output
  - specific access modes for reductions

• **Compiled thanks to KaCC**
  - Based on the ROSE compiling framework
  - Support for C/C++ applications
  - #pragma kaapi annotations
OpenMP-like example of data-flow programming with libKOMP

```c
void data_flow_example (void)
{
    type x, y;

    #pragma kaapi parallel
    {
        #pragma kaapi task write(x)
        compute_smthg (&x); /* Task A */
        #pragma kaapi task write(y)
        compute_smthg (&y); /* Task B */

        #pragma kaapi task read(x)
        process_results (x); /* Task C */
        #pragma kaapi task read(y)
        process_results (y); /* Task D */
    }
}
```

Valid execution orders:
- A, B, C, D
- A, B, D, C
- B, A, C, D
- B, A, D, C
- B, D, A, C (*)
- A, C, B, D (*)
*invalid on the taskwait version

Invalid execution orders:
- C before A
- D before B
Enhance the EUROPLEXUS application with data-flow tasks

• The EUROPLEXUS application
  ‣ Industrial code, originally developed by CEA
  ‣ Analyzes the propagation of flow variations in presence of obstacles
  ‣ Sparse Cholesky factorization on a skyline matrix

• Two different versions of the application
  ‣ Standard OpenMP 3.0 version
    ○ Independent tasks with explicit synchronizations
  ‣ libKOMP-powered data-flow version
    ○ Expressing variable access modes
    ○ Let the runtime system decide whether a task is ready or not
Evaluation of independent tasks vs Data-flow tasks on the Europlexus application

![Graph showing speedup vs core count for different parallelization methods. The graph compares the ideal performance, libKOMP, and OpenMP. The x-axis represents the core count, and the y-axis represents the speedup (Tp/Tseq).]