The multicore and memory revolution
- ISA leak …
- Plethora of architectures
  - Heterogeneity
  - Memory hierarchies

Complexity +
+ variability =
= Divergence …
- … between our mental models and actual system behavior
The programming revolution

An age changing revolution

- From the latency age …
  - I need something … I need it now !!!
  - Performance dominated by latency in a broad sense
    - At all levels: sequential and parallel
    - Memory and communication, control flow, synchronizations

- …to the throughput age
  - Ability to instantiate “lots” of work and avoid stalling for specific requests
    - I need this and this and that … and as long as it keeps coming I am ok
    - (Much broader interpretation than just GPU computing !!!)
  - Performance dominated by overall availability/balance of resources
Vision in the programming revolution

Need to decouple again

Applications

PM: High-level, clean, abstract interface

Power to the runtime

ISA / API

Application logic

Arch. independent

General purpose
Task based
Single address space

“Programming language”

“Reuse” architectural ideas under new constraints
Co’s opportunities and challenges

“Co-design”: features in one level to support/adapt to other levels
- Holistic design/optimization
  - Multilevel
  - Multidirectional
- Fundamentals
- Dynamic

“Co-ordination” between dynamic resource management levels

“Co-nvergence” of sectors
- HPC. Embedded, Big data/analytics
Vision in the programming/architecture revolution

Back to the future …

Applications
PM: High-level, clean, abstract interface

Architecture aware runtime
Runtime aware architecture

Application logic
Arch. independent

Programmer
Engine interface

Flexibility in execution engine design

PM: High-level, clean, abstract interface

Application logic
Arch. independent

Programmer
Engine interface

Flexibility in execution engine design

PM: High-level, clean, abstract interface

Application logic
Arch. independent

Programmer
Engine interface

Flexibility in execution engine design
The StarSs family of programming models

**Key concept**
- **Sequential task based** program on **single address/name space** + **directionality annotations**
- Happens to execute parallel: Automatic run time computation of dependencies between tasks

**Differentiation of StarSs**
- Dependences: Tasks instantiated but not ready. Order IS defined
  - **Lookahead**
    - Avoid stalling the main control flow when a computation depending on previous tasks is reached
    - Possibility to “see” the future searching for further potential concurrency
  - **Locality aware**
    - Without defining new concepts
- **Homogenizing heterogeneity**
  - Device specific tasks but homogeneous program logic
History / Strategy

- PERMPAR ~1994
- GridSs ~2002
- CellSs ~2006
- SMPSs V1 ~2007
- SMPSs V2 ~2009
- StarSs ~2008
- GPUSs ~2009
- COMPSs ServiceSs ~2010
- COMPSs ServiceSs PyCOMPSs ~2013
- OmpSs ~2008
- NANOS ~1996
- DDT @ Parascope ~1992

Forerunner of OpenMP

OpenMP 3.0 4.0
2008 2013
OmpSs in one slide

Minimalist set of concepts …
– … ”extending” OpenMP
– … relaxing StarSs funtional model

#pragma omp task [ in (array_spec...)][ out (...)] [inout (...)] \ 
[ concurrent (...)][ commutative(...) ] [ priority(P) ] [ label(...) ] \ 
[ shared(...)][private(...)][firstprivate(...)][default(...)][untied][final][if (expression)] 
[reduction(identifier : list)]
{code block or function}

#pragma omp taskwait [ on (...) ][noflush]

#pragma omp for [ shared(...)][private(...)][firstprivate(...)][schedule_clause] 
{for_loop}

#pragma omp taskloop [grainsize(...) ]... [nogroup] [ in (...) ] ... 
{for_loop}

#pragma omp target device ({ smp | opencl | cuda }) \ 
[ implements ( function_name )] \ 
[ copy_deps | no_copy_deps ] [ copy_in ( array_spec ,...)] [ copy_out (...)] [ copy_inout (...)] \ 
[ndrange (dim, …)] [shmem(...) ]
Adapting concurrency to dynamic environment

- Sequential file processing
- Automatically achieve asynchronous I/O

```c
typedef struct {int size, char buff[MAXSIZE]} buf_t;

buf_t *p[NBUF];
int j=0, total_records=0;

int main()
{
    ...
    while(!end_trace) {
        buf_t **pb=p[j%NBUF]; j++;
        #pragma omp task inout(infile) out(*pb, end_trace) \ 
        priority(10)
        { *pb = malloc(sizeof(buf_t));
          Read (infile, *pb, &end_trace);
        }

        #pragma omp task inout(*pb)
        Process (*pb);

        #pragma omp task inout(outfile, *pb, total_records)\ 
        priority(10)
        { Write (outfile, *pb, &records);
          total_records += records;
          free (*pb);
        }

        #pragma omp taskwait on (&end_trace)
    }

    ...
}
```
PARSEC benchmark ported to OmpSs

**Initial port from pthreads to OmpSs and optimization**

**Bodytrack**

**Ferret**

---

"Direct"

0-10

"optimized"

0-250

---

D. Chasapis et al, “Exploring the Impact of Task Parallelism Beyond the HPC Domain”
Task reductions

- While-loops, recursions
- On its way to OpenMP 5.0


Task reductions

Array reductions
- Typical pattern: reductions on large arrays with indirection

Implementation
- Privatization becomes inefficient when scaling cores and data size
- Atomics can introduce significant overhead
- PIBOR Proposal: Privatization with in-lined block-ordered reductions
  - Save footprint
  - Trade processor cycles for locality

```c++
for (auto t : tasks){
    #pragma task
    reduction (+:v[0:size]) \ private (j)
    for (auto i : taskIters) {
        j= f(i);
        v[j] += expression;
    }
}
#pragma taskwait
```

Ciesko, J., et al. “Boosting Irregular Array Reductions through In-lined Block-ordering on Fast Processors”, HPEC15
Programmability improvements

Frequent pattern:
- Different instantiations of same task requiring different number of dependences imply code replication (peeling, switch,…)
  - E.g. number of neighbors in a domain decomposition

Multidependences
- Syntax to specify a dynamic number of directionality clauses

```c
#pragma omp task in(v[neigh.n[j]], j=0:neigh.size())
//task code
```

May require the pre-computation of some “connectivity” information
- Often already there in many codes
- Frequently an invariant information (not required by the proposal but if it is the case the overhead of its computation is amortized over multiple uses)
Homogenizing Heterogeneity

ISA heterogeneity

Single address space program ... executes in several non coherent address spaces

- Copy clauses:
  - ensure sequentially consistent copy accessible in the address space where task is going to be executed
  - Requires precise specification of data accessed (e.g. array sections)
- Runtime offloads data and computation and manages consistency

Kernel based programming

- Separation of iteration space identification and loop body

```
#pragma omp target device ( { smp | opencl | cuda } ) \ 
[ copy_deps | no_copy_deps ] [ copy_in ( array_spec ,...)] [ copy_out (...)] [ copy_inout (...)] } \ 
[ implements ( function_name )] \ 
[ shmemb(... ) ] \ 
[ ndrange (dim, g_array, l_array)]
```

```
#pragma omp taskwait [ on (...) ][noflush]
```
#pragma omp target device(opencl) nndrange(1,size,128) copy_deps implements (calculate_forces)
#pragma omp task out([size] out) in([npart] part)
__kernel void calculate_force_opencl(int size, float time, int npart, __global Part* part,
__global Part* out, int gid);

#pragma omp target device(cuda) nndrange(1,size,128) copy_deps implements (calculate_forces)
#pragma omp task out([size] out) in([npart] part)
__global__ void calculate_force_cuda(int size, float time, int npar, Part* part, Particle *out, int gid);

#pragma omp target device(smp) copy_deps
#pragma omp task out([size] out) in([npart] part)
void calculate_forces(int size, float time, int npart, Part* part, Particle *out, int gid);

void Particle_array_calculate_forces(Particle* input, Particle *output, int npart, float time) {
    for (int i = 0; i < npart; i += BS )
        calculate_forces(BS, time, npart, input, &output[i], i);
}
MACC (Mercurium ACcelerator Compiler)

“OpenMP 4.0 accelerator directives” compiler
- Generates OmpSs code + CUDA kernels (for Intel & Power8 + GPUs)
- Propose clauses that improve kernel performance

Extended semantics
- Change in mentality … minor details make a difference
- Dynamic parallelism

<table>
<thead>
<tr>
<th>Type of device</th>
<th>Ensure availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>OmpSs</td>
<td></td>
</tr>
<tr>
<td>for (...)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>#pragma omp target device(acc) copy_deps</td>
<td></td>
</tr>
<tr>
<td>#pragma omp task inout(x[beg:end])</td>
<td></td>
</tr>
<tr>
<td>#pragma omp teams distribute parallel for</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; ..Computation.. &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specific device</th>
<th>DO transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenMP 4.0</td>
<td></td>
</tr>
<tr>
<td>for (...)</td>
<td></td>
</tr>
<tr>
<td>{</td>
<td></td>
</tr>
<tr>
<td>int dev_id = i % omp_get_num_devices();</td>
<td></td>
</tr>
<tr>
<td>#pragma omp task</td>
<td></td>
</tr>
<tr>
<td>#pragma omp target device(dev_id) map(tofrom: x[beg:end])</td>
<td></td>
</tr>
<tr>
<td>#pragma omp teams distribute parallel for</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; ..Computation.. &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

G. Ozen et al, “On the roles of the programmer, the compiler and the runtime system when facing accelerators in OpenMP 4.0” IWOMP 2014

G. Ozen et al, “Multiple Target Work-sharing support for the OpenMP Accelerator Model” submitted
Integration of MPI and OmpSs

Taskifying MPI calls

Opportunity
- Overlap between phases
  - Grid and frequency domain
- Provide laxity for communications
  - Tolerate poorer communication
- Migrate load balance issue
  - Flexibility for DLB
- Huge flexibility for changing behavior with minimal syntactic changes

IFS weather code kernel. ECMRWF

Hybrid Amdahl’s law

- A fairly “gad message” for programmers
- Significant non parallelized part
  - MPI calls + pack/unpack

MPI + OmpSs: Hope for lazy programmers
```c
MPI_Comm comm_slaves, comm_workers;

void processShot(int l, shot_handler_t* shot) {
    read_shot(shot); // Read shot from GPFS
    int psize = shot->size / 4;
    for(int i=0; i<4; i++) {
        float* data = &shot->data[i*psize];
        #pragma omp task input(data[0:psize]) onto(comm_workers, i)
        processShotSlice(data, psze);
    }
    #pragma omp taskwait
}

int main(int argc, const char* argv[]) {
    int n_slaves = 256, nshots = n_slaves * 32;
    shot_handler_t shots[nshots];
    // Master nodes allocate slaves 256 slaves
depth_loader_alloc(MPI_COMM_SELF, 16, 16, &comm_slaves);

    for(int i=0; i<n_slaves; i++) {
        // Each slave allocates 4 workers
        #pragma omp task onto(comm_slaves, i)
depth_loader_alloc(MPI_COMM_SELF, 4, 1, &comm_workers);
    }

    for(int i=0; i<nshots; i++) {
        #pragma omp task input(shots[i]) onto(comm_slaves)
        processShot(&shots[i]);
        merge_and_save_shot(&shots[i]); // GPFS I/O intensive
    }
    #pragma omp taskwait
    [...]
DSLs

A. Fernandez et al, “A Data Flow Language to Develop High Performance Computing DSLs” Submitted

val c = Cartesian(12.5, 25.0, 37.5)
val temp = Unknown(c)
val cond = Dirichlet(lowXZ of c, temp, 400)
val hv = Vector(0.5, 0.5, 0.5)
val pre = PreProcess(nsteps = 10000, deltaT = 0.25, h = hv)(cond)
solve(pre) equation (0.15 * lapla(temp) - dt(temp)) to "diffusion"
OmpSs

Other features and characteristics

- Contiguous / strided dependence detection and data management
- Nesting/recursion
- Multiple implementations for a given task
- CUDA, OpenCL
- Cluster
- Dynamic load balance between MPI processes in one node
- Resilience

Commitment

- Continuous development and use since 2004
- Pushing ideas into the OpenMP standard
  - Dependences $\rightarrow$ v 4.0
  - Priority $\rightarrow$ v 4.5
- Developer Positioning: efforts in OmpSs will not be lost.
COMPILER AND RUNTIME
Mercurium

- Source-to-source compiler (supports OpenMP and OmpSs extensions)
- Recognize pragmas and transforms original program to call Nanox++
- Supports Fortran, C and C++ languages (backends: gcc, icc, nvcc, …)
- Supports complex scenarios
  - Ex: Single program with MPI, OpenMP, CUDA and OpenCL kernels

http://pm.bsc.es
Nanos++

- Common execution runtime (C, C++ and Fortran)
- Task creation, dependence management, resilience, …
- Task scheduling (FIFO, DF, BF, Cilk, Priority, Socket, affinity, …)
- Data management: Unified directory/cache architecture
  - Transparently manages separate address spaces (host, device, cluster)…
  - … and data transfer between them
- Target specific features

The NANOS++ Runtime

http://pm.bsc.es
Criticality-awareness in heterogeneous architectures

- Heterogeneous multicores
  - ARM big.LITTLE 4 A-15@2GHz; 4A-7@1.4GHz
  - Tasksim simulator: 16-256 cores; 2-4x

- Runtime approximation of critical path
  - Implementable, small overhead that pay off
  - Approximation is enough

- Higher benefits the more cores, the more big cores, the higher performance ratio

---

FPGAs

Just another heterogeneous device

Experiments @ Zynq

```c
#pragma omp target device(smp, fpga) copy_deps
#pragma omp task inline
void MxM_Kernel(T a[Dim][Dim], T b[Dim][Dim], T out[Dim][Dim]){
    #pragma HLS inline
    #pragma HLS array_partition variable=a block factor=Dim/2 dim=2
    #pragma HLS array_partition variable=b block factor=Dim/2 dim=1
    for (int ia = 0; ia < Dim; ++ia)
        for (int ib = 0; ib < Dim; ++ib)
        {
            #pragma HLS pipeline
            T sum = out[ia][ib];
            for (int id = 0; id < Dim; ++id)
                sum += a[ia][id] * b[id][ib];
            out[ia][ib] = sum;
```
Managing separate address spaces

**OmpSs @ Cluster**
- Directory @ master
- A software cache @ device manages its individual address space:
  - Manages local space at device (logical and physical)
  - Translate address @ main address space → device address
- Implements transfers
  - Packing if needed
  - Device/network specific transfer APIs (i.e. GASNet, CUDA copies, MPI, …)

**A productive alternative for programs requiring very large footprints**
- Omics
- Graph analytics

J. Bueno et al, “Productive Programming of GPU Clusters with OmpSs”, IPDPS2012
J. Bueno et al, “Implementing OmpSs Support for Regions of Data in Architectures with Multiple Address Spaces”, ICS 2013
Device management mechanisms

**Improvements in runtime mechanisms**
- Use of **multiple streams**
- High asynchrony and overlap (transfers and kernels)
- Overlap kernels
- Take overheads out of the critical path

**Improvement in schedulers**
- Late binding of locality aware decisions
- Propagate priorities

---

Locality aware scheduling
- Affinity to core/node/device can be computed based on pragmas and knowledge of where was data
- Following dependences reduces data movement
- Interaction between locality and load balance (work-stealing)

Some “reasonable” criteria
- Task instantiation order is typically a fair criteria
- Honor previous scheduling decisions when using nesting
  - Ensure a minimum amount of resources
  - Prioritize continuation of a father task in a taskwait when synchronization fulfilled
Dynamic Load Balancing

- Dynamically shift cores between processes in node
  - Enabled by application malleability (task based)
  - Runtime in control (sees what happens and the future)
  - Would greatly benefit from OS support (handoff scheduling, fast context switching)

“LeWI: A Runtime Balancing Algorithm for Nested Parallelism”. M.Garcia et al. ICPP09
- Real time
- Memory association changes
- Runtime reengineering
- OMPT
- Tareador
- Debugging - Temanejo @ HLRS
- …
CONCLUSION
The importance of …

Language
- Transport / integrate information
  - Experience $\rightarrow$ Good “predictor”
- Structures mind

Details
- Very “simple/small/subtle” differences $\rightarrow$ very important impact
What is important?

- Asynchrony/dataflow
- Data access based specification and runtime awareness
- Malleability
- Not to fly blind

Dynamic
The real parallel programming revolution …

... is in the mindset of programmers

... is to rely on the (runtime) – system

We all have to learn about data flow, overall potential concurrency, malleability, ...
In times of confusion, when multicores broke our quiet and steady growth, we entered a battle no one can win.

StarSs

Or may be yes?
THANKS