Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Hybrid task based programming in the HPC domain

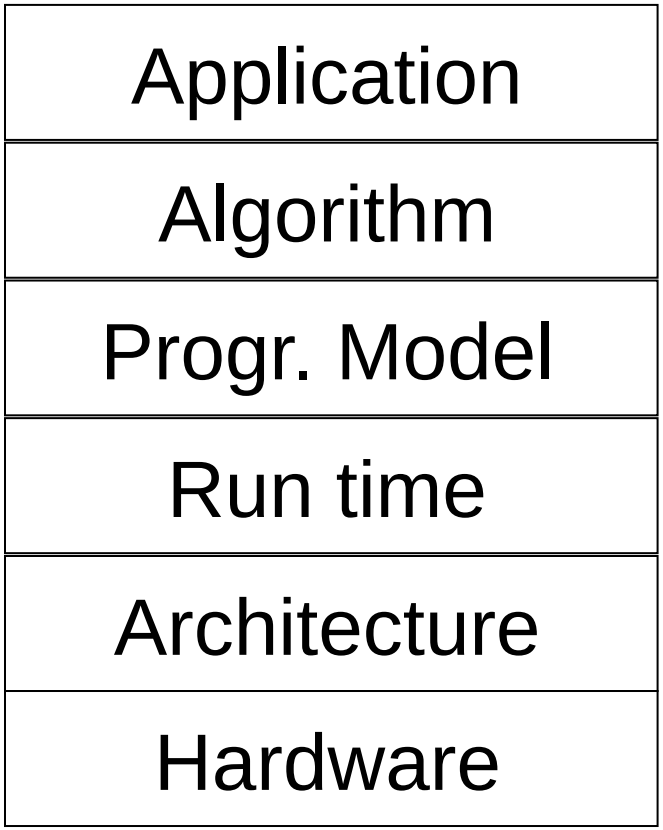
Prof. Jesús Labarta

BSC & UPC

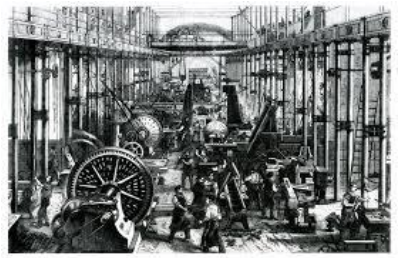
ACM Summer school

Barcelona, July 22nd 2019

Where do you live ?



Transitions



The programming 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
    - In a broad sense
  - Performance dominated by overall availability/balance of resources

Practices and opportunities …

- … to transition to the throughput age …
- …with hybrid MPI+ OpenMP tasks …
- … productively

Outline

- Vision
- Basics of programming models
  - OpenMP, OmpSs
  - MPI
  - Hybrid
- Performance insight
  - Don’t mask pour symptoms
- Hybrid MPI+ OpenMP/OmpSs
  - First example
  - Taskifying communications
  - Malleability
  - Reductions
  - Hierarchical overdecomposition
- Concluding remarks

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

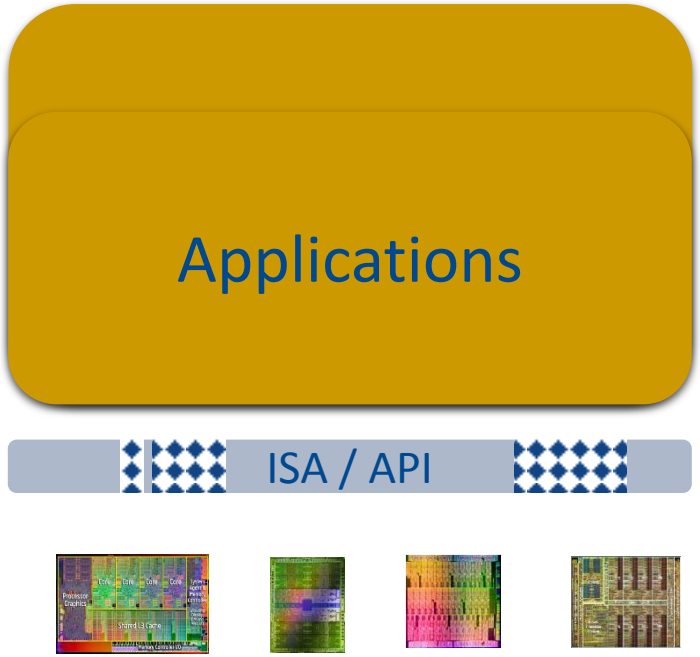
EXCELENCIA SEVERO OCHOA

A personal vision

Vision

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

The power wall made us go multicore and the ISA interface to leak
→ our world is shaking

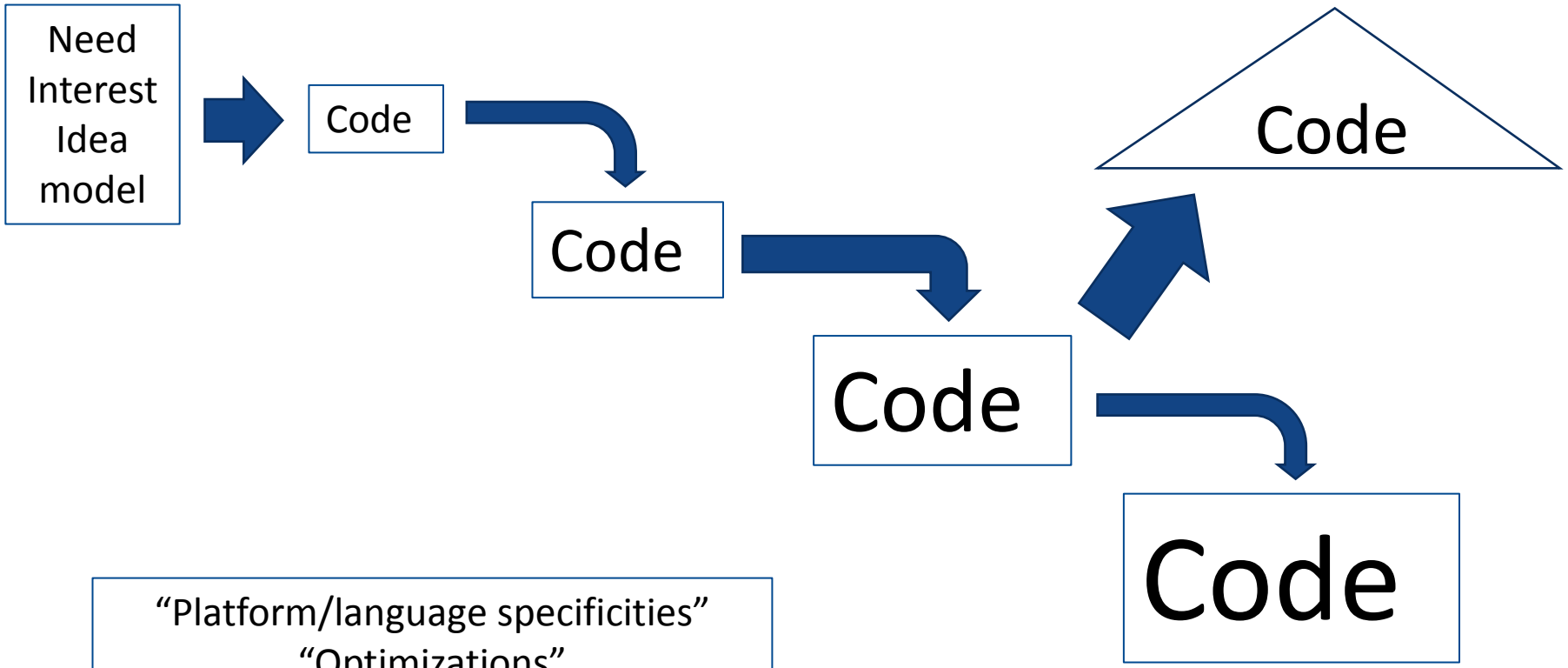


- Complexity + variability = Divergence
  - Between our mental models and actual system behavior

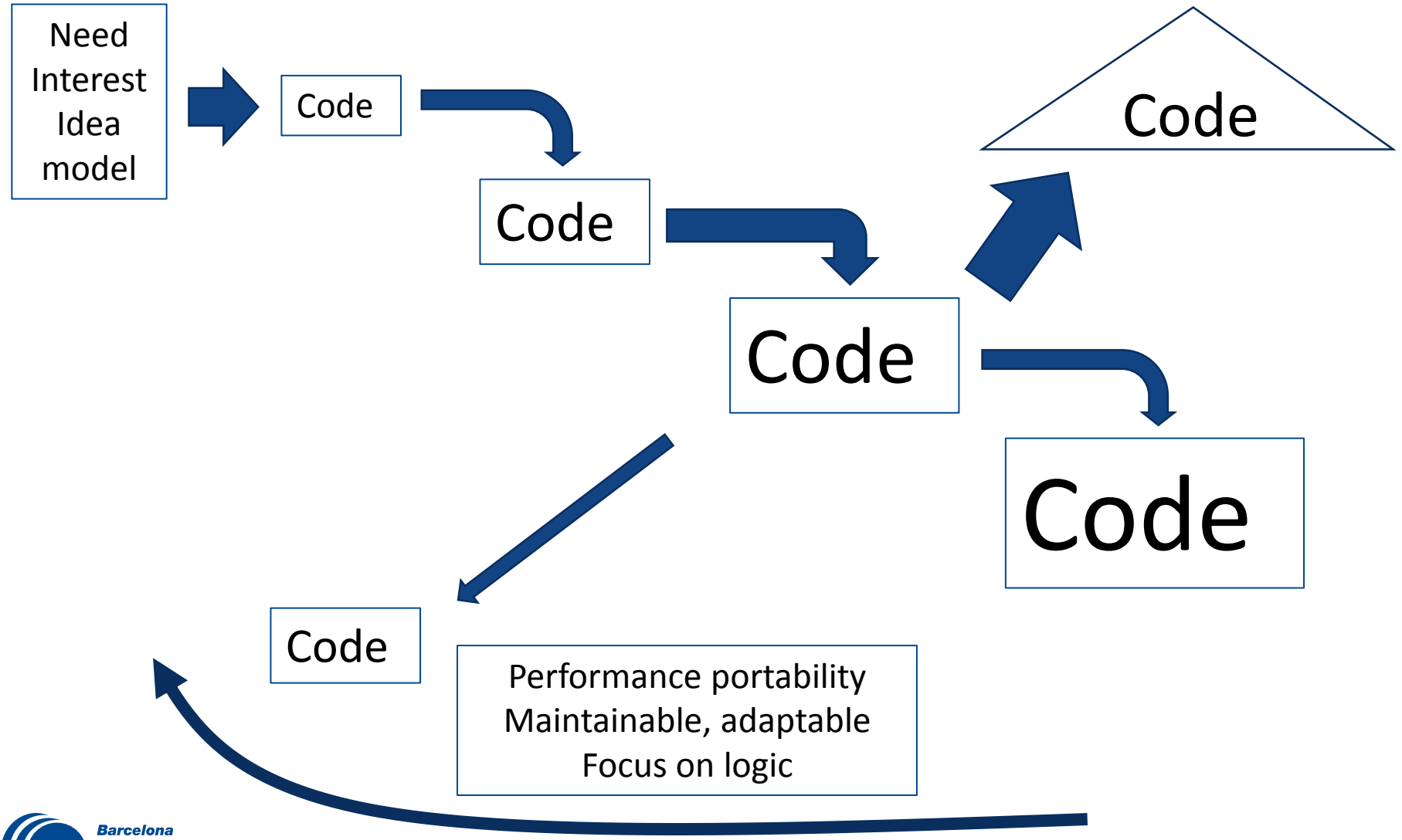
What programmers need ?

HOPE !!!

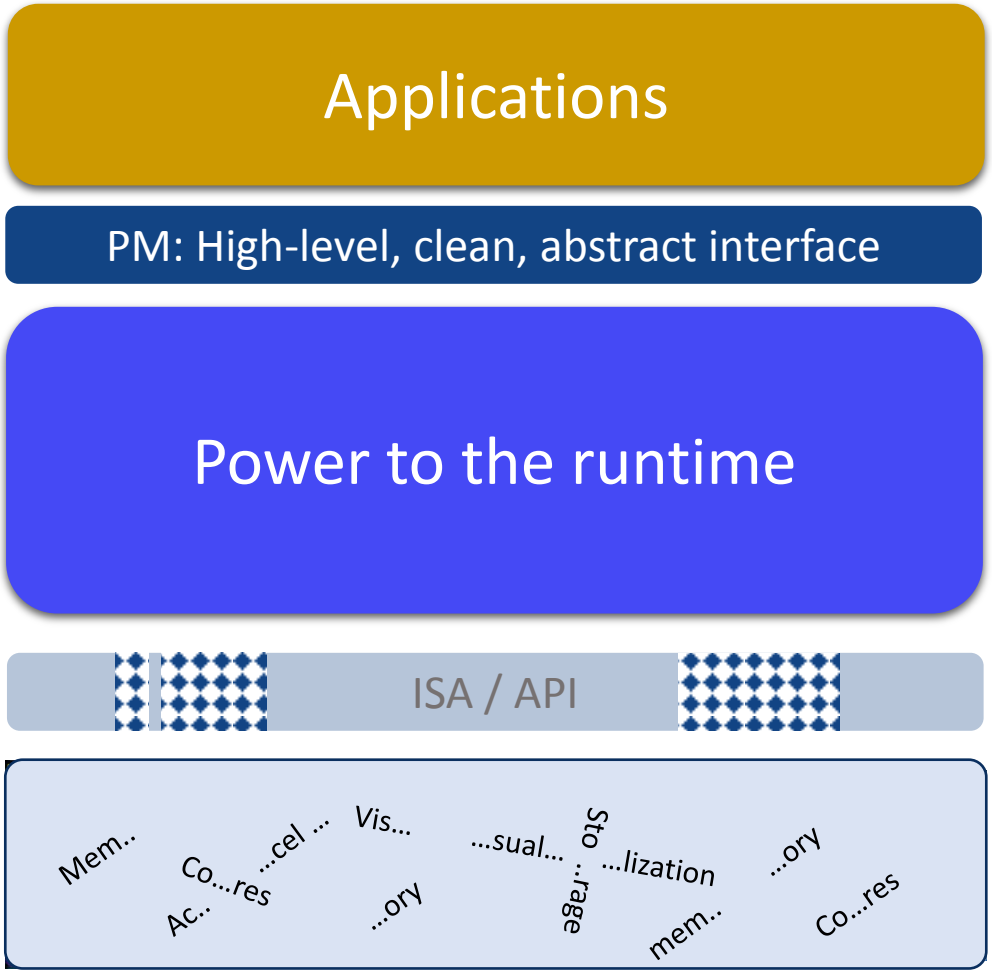
Code lifetime cycle



There is HOPE !!!



Vision in the programming revolution



General purpose

Task & data based

Forget about resources

Decouple:
Minimal & sufficient permeability?

Intelligence
&
Resource management

"Reuse & expand" old architectural ideas under new constraints

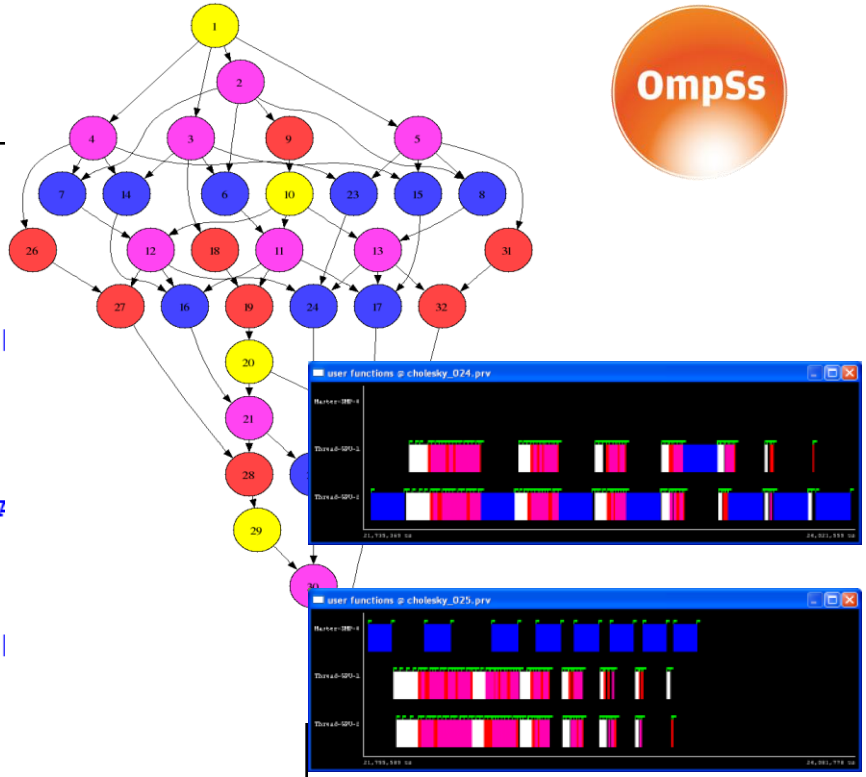
The StarSs family of programming models

- **A sequential task based** program …
- … on a **single address/name space …**
- **… with directionality annotations …**
- …that happens to **execute in parallel**

Integrate concurrency and data

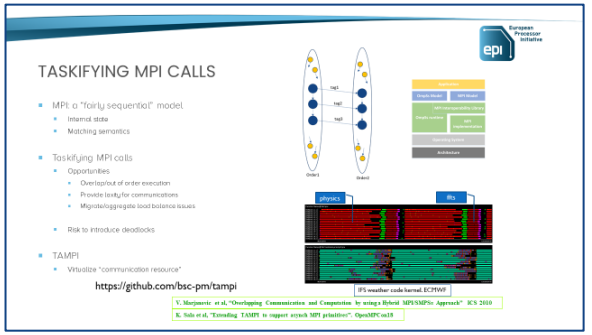
- Single mechanism
  - Concurrency:
    - Dependences built from data accesses
    - Lookahead: About instantiating work
  - Locality & data management
    - From data accesses

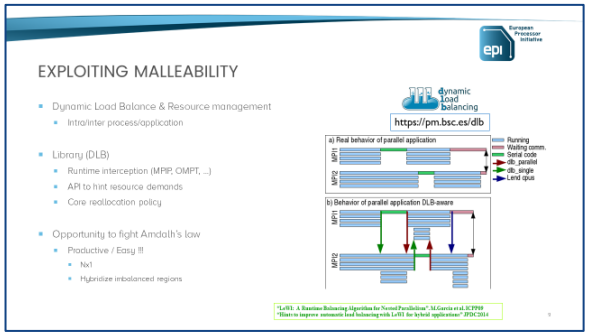
```
void Cholesky(int NT, float *A[NT][NT] ) {
   for (int k=0; k<NT; k++) {
      #pragma omp task inout ([TS][TS](A[k][k]))
      spotrf (A[k][k], TS) ;
      for (int i=k+1; i<NT; i++) {
        #pragma omp task in([TS][TS](A[k][k])) inout ([TS]
        strsm (A[k][k], A[k][i], TS);
      }
      for (int i=k+1; i<NT; i++) {
        for (j=k+1; j<i; j++) {
          #pragma omp task in([TS][TS](A[k][i]), [TS][TS](
                           inout ([TS][TS](*A[j][i]))
          sgemm( A[k][i], A[k][j], A[j][i], TS);
        }
        #pragma omp task in ([TS][TS](A[k][i])) inout([TS]
        ssyrk (A[k][i], A[i][i], TS);
      }
   }
}
```



Important topics/practices

- Regions
- Nesting
- Taskloops + dependences
- Taskify communications: MPI Interoperability
- Malleability
- Homogenize Heterogeneity
- Hierarchical “acceleration” / Hierarchical overdecomposition
- The osmotic porosity (hints, overheads,…)
- Beware of reductions on large arrays
- Memory management & Locality
- Beyond the node?
- Don´t mask your symptoms





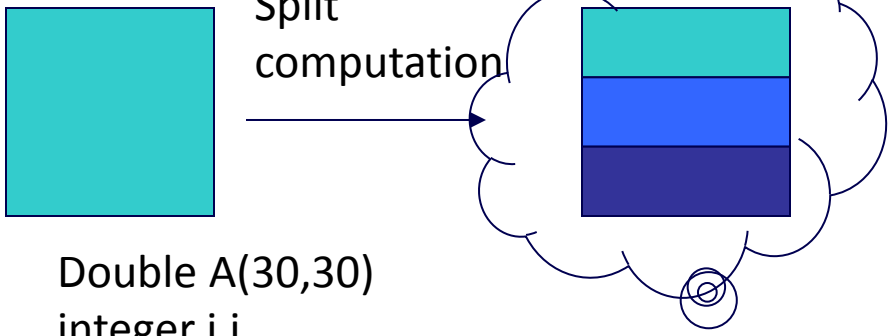
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

OpenMP and OmpSs

OpenMP

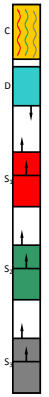
Sequential




```
Double A(30,30)
integer i,j

Do j = 1,30
  Do i =1,30
    A(i,j) = 2* A(i,j)
    A(i,j) = j* A(i,j)
    A(i,j) = i* A(i,j)
```

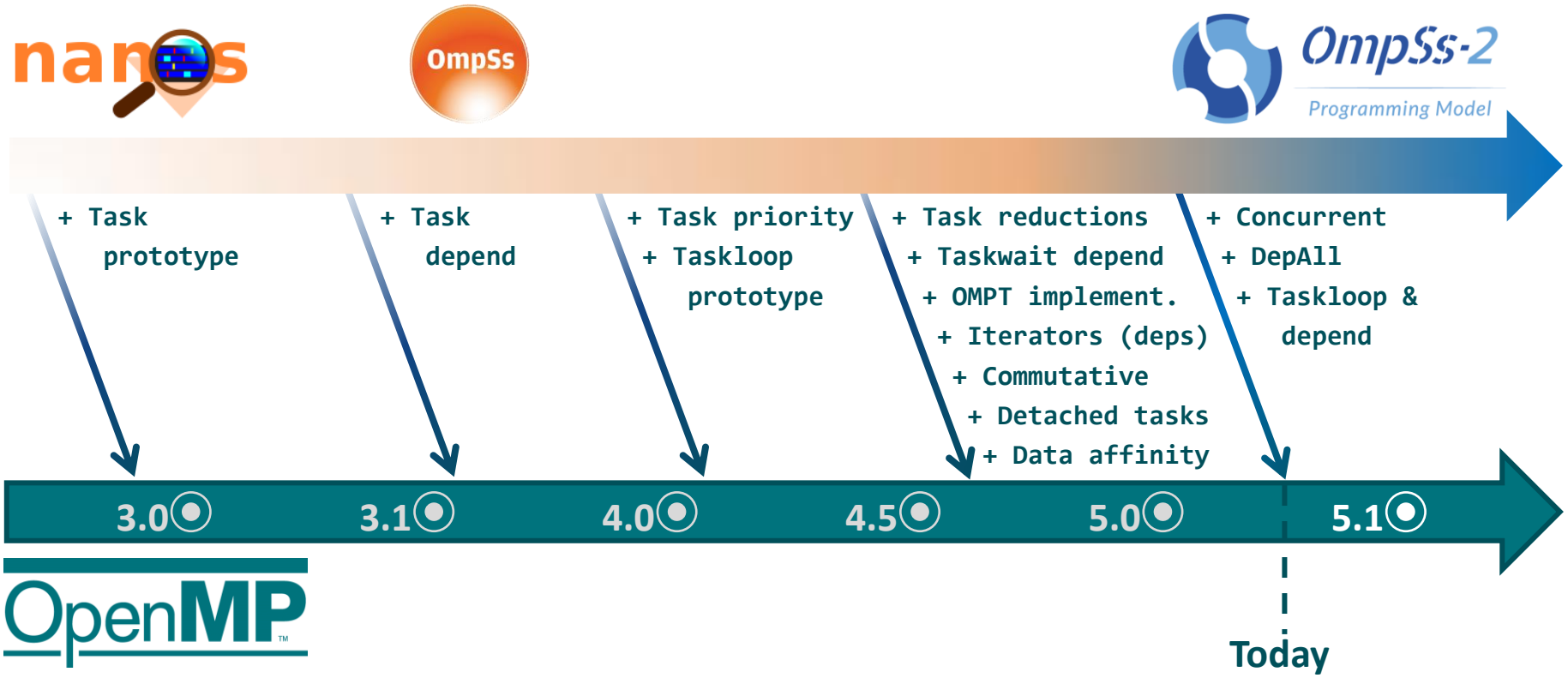
```
Double A(30,30)
interger i,j

  Do j = 1,30
C$OMP  PARALLEL DO
C$OMP  PRIVATE (i)
    Do i =1,30
      A(i,j) = 2* A(i,j)
      A(i,j) = j* A(i,j)
      A(i,j) = i* A(i,j)
```

Very
Fork - joinish



OmpSs

- A forerunner for OpenMP Tasking



OmpSs in one slide (201?)

Color code: OpenMP, influenced OpenMP, pushing, not yet

- Minimalist set of concepts …

```
#pragma omp task [ in (array_spec, l_values...)] [ out (...)] [ inout (…, v[neigh[j]], j=0;n)]) \
                 [ concurrent (…)] [commutative(...)] [ priority(P) ] [ label(...) ] \
                 [ shared(...)][private(...)][firstprivate(...)][default(...)][untied] \
                 [final(expr)][if (expression)] \
                 [reduction(identifier : list)] \
                 [resources(…)]
 {code block or function}
```

```
#pragma omp taskwait [ { in | out | inout } (...) ] [noflush]
```

```
#pragma omp taskloop  [grainsize(…) ] [num_tasks(…) [nogroup]   [ in (...)] [reduction(identifier : list)]
  {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(...) ]
```

A. Duran, et al, "Extending the OpenMP Tasking Model to Allow Dependent Tasks" IWOMP 2008, LNCS & IJPP

E. Ayguade, et al, "A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures" IWOMP 2009 & IJPP

OmpSs in one slide (2019)

Color code: OpenMP, influenced OpenMP, pushing, not yet

- Minimalist set of concepts …

```
#pragma omp task [ in (array_spec, l_values...)] [ out (...)] [ inout (…, v[neigh[j]], j=0;n)]) \
                 [ concurrent (…)] [commutative(...)] [ priority(P) ] [ label(...) ] \
                 [ shared(...)][private(...)][firstprivate(...)][default(...)][untied] \
                 [final(expr)][if (expression)] \
                 [reduction(identifier : list)] \
                 [resources(…)]
 {code block or function}
```

```
#pragma omp taskwait [ { in | out | inout } (...) ] [noflush]
```

```
#pragma omp taskloop  [grainsize(…) ] [num_tasks(…) [nogroup]   [ in (...)] [reduction(identifier : list)]
  {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(...) ]
```

A. Duran, et al, "Extending the OpenMP Tasking Model to Allow Dependent Tasks" IWOMP 2008, LNCS & IJPP

E. Ayguade, et al, "A Proposal to Extend the OpenMP Tasking Model for Heterogeneous Architectures" IWOMP 2009 & IJPP

OpenMP vs OmpSs

- Execution model
  - To thread or not to thread
    - Thread == Resource !!!!
  - OpenMP:
    - Parallel
      - team of threads
      - All of them must be involved in computation.
      - Fork join synchronization context
  - OmpSs:
    - Purely task based
      - Computations and their ordering relations/restrictions
    - Pool of threads not part of the model

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

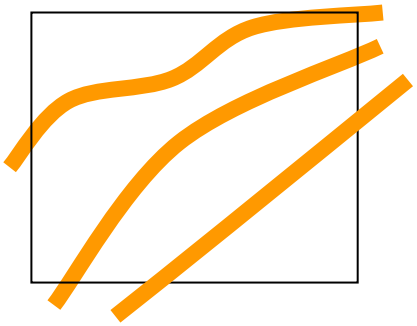
OmpSs-2

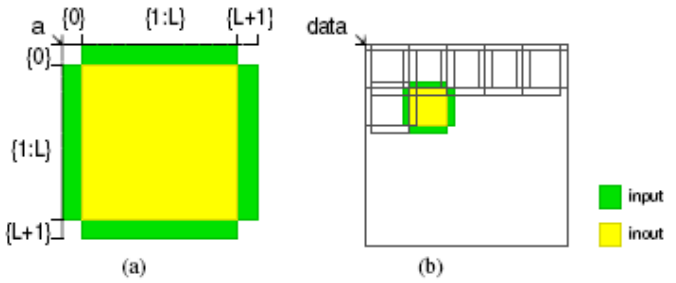
OmpSs-2 – key features

- Region dependencies
  - Precise overlap detection of overlap between N-dimensional region specifications
- Nesting
  - Runtime flattening of dependences across nested tasks
  - Weak dependencies
- MPI + OpenMP interoperability
  - Virtualized cores for unlimited number of blocked tasks in MPI (TAMPI)

Regions

- Precise nD subarray accesses
  - “Complex” analysis but …
- Enabler for …
  - Recursion
  - Flexible nesting
  - Taskloop dependences
  - Data management
    - locality
    - layout



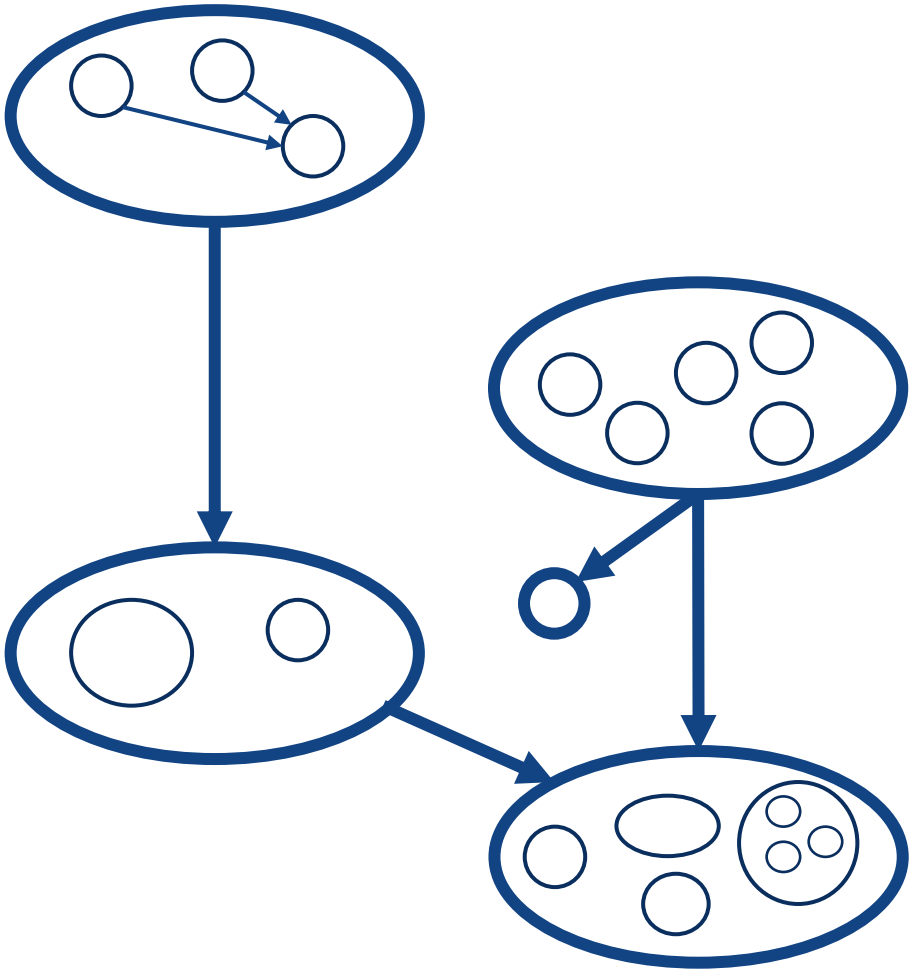



```
void gs (float A[(NB+2)*BS][(NB+2)*BS])
{
  int it,i,j;

  for (it=0; it<NITERS; it++)
    for (i=0; i<N-2; i+=BS)
      for (j=0; j<N-2; j+=BS)
        gs_tile(&A[i][j]);
}
```

```
#pragma oss task \
  in(A[0][1;BS], A[BS+1][1;BS], \
       A[1;BS][0], A[1:BS][BS+1]) \
  inout(A[1;BS][1;BS])
void gs_tile (float A[N][N])
{
  for (int i=1; i <= BS; i++)
    for (int j=1; j <= BS; j++)
      A[i][j] = 0.2*(A[i][j] + A[i-1][j] +
                A[i+1][j] + A[i][j-1] +
                A[i][j+1]);
}
```

Nesting

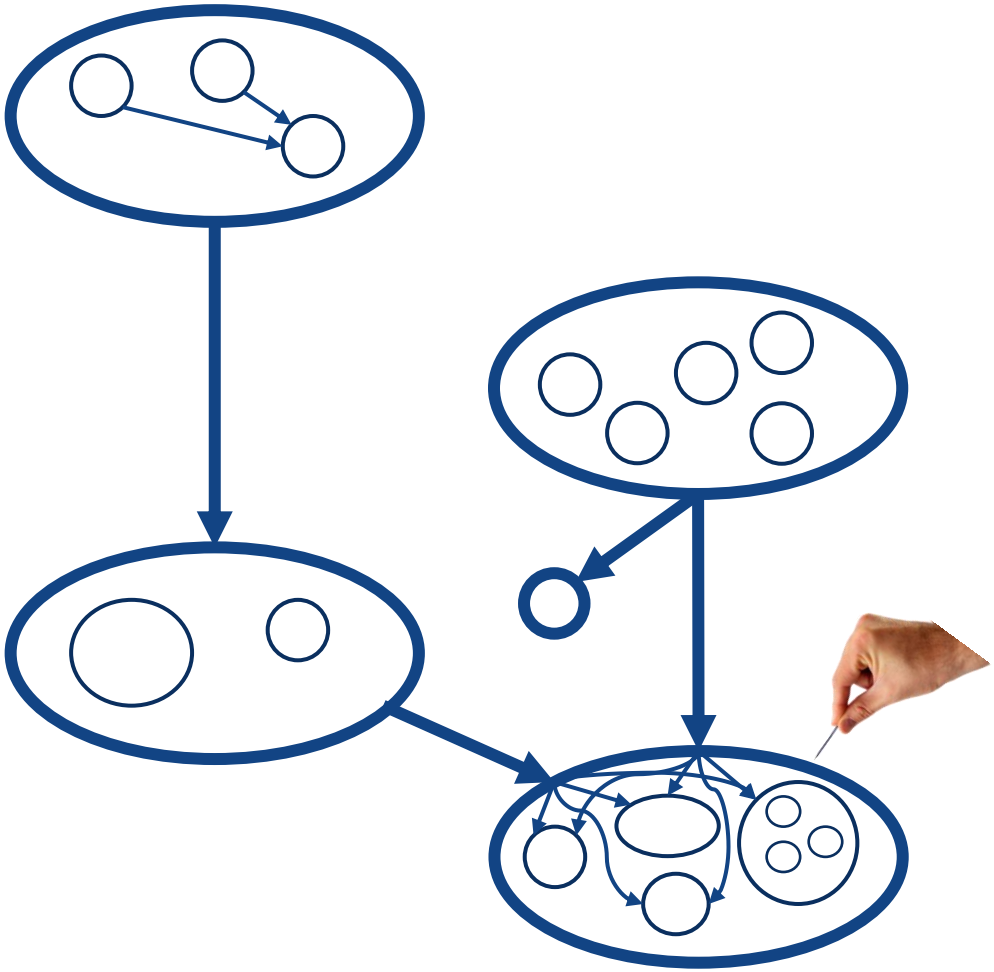
- Top down
  - Every level contributes
- Flattening dependence graph
  - Increase concurrency
  - Take out runtime overhead from critical path
- Granularity control
  - final clauses, runtime



J. M. Perez, et all, "Improving the Integration of Task Nesting and Dependencies in OpenMP" IDPS 2017

Nesting

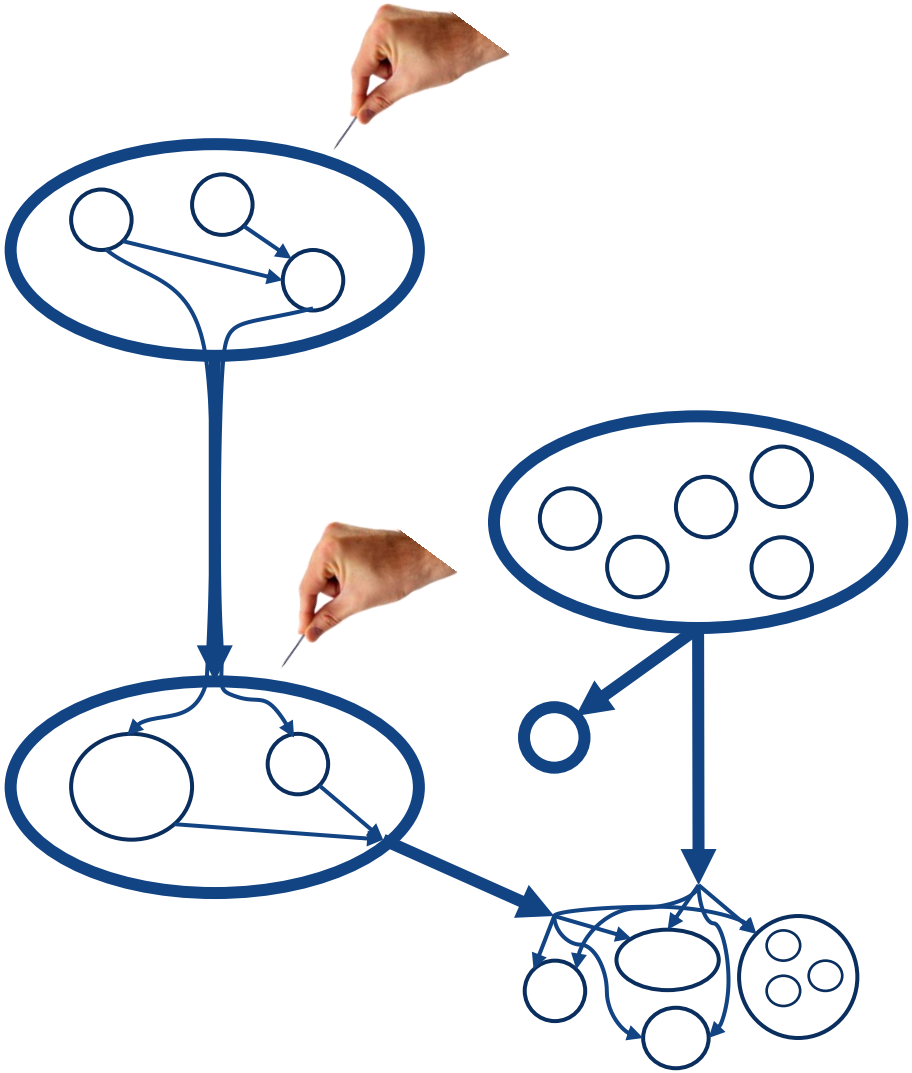
- Top down
  - Every level contributes
- Flattening dependence graph
  - Increase concurrency
  - Take out runtime overhead from critical path
- Granularity control
  - final clauses, runtime



J. M. Perez, et all, "Improving the Integration of Task Nesting and Dependencies in OpenMP" IPDPS 2017

Nesting

- Top down
  - Every level contributes
- Flattening dependence graph
  - Increase concurrency
  - Take out runtime overhead from critical path
- Granularity control
  - final clauses, runtime



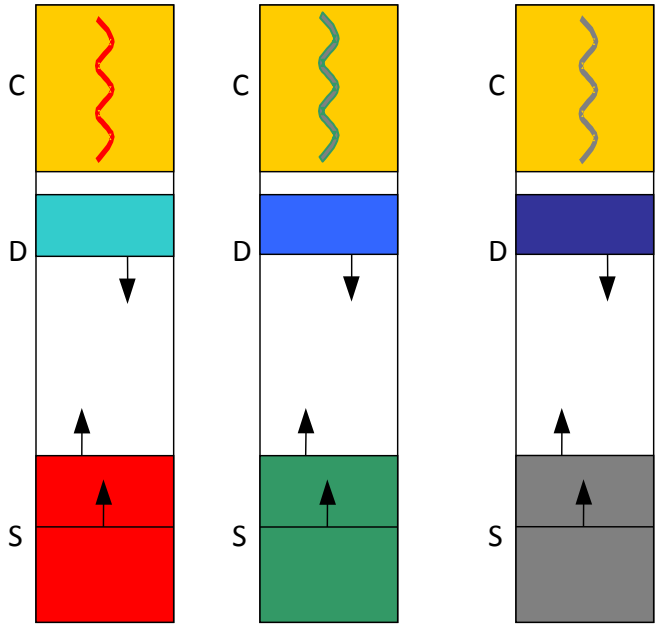
J. M. Perez, et all, "Improving the Integration of Task Nesting and Dependencies in OpenMP" IDPS 2017

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

MPI

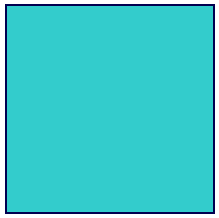
MPI



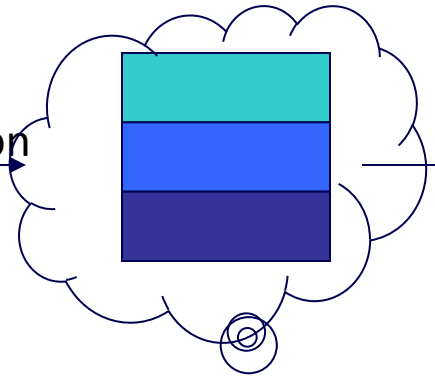

```
Includes

Global variables

main () {
   my_part = f( who am I)

   Compute my part
   Communicate if needed

}
```

MPI

Sequential



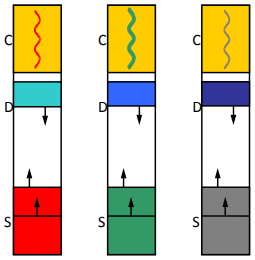
Data distribution



Distributed Memory

```
Double A(30,30)
integer i,j

Do j = 1,30
  Do i =1,30
    A(i,j) = 2* A(i,j)
    A(i,j) = j* A(i,j)
    A(i,j) = i* A(i,j)
```

```
Double A(10,30)

myrank = quien_soy()
Do j = 1,30
  Do li =1,10
    i=li+myrank*10
    A(li,j) = 2* A(li,j)
    A(li,j) = j* A(li,j)
    A(li,j) = i* A(li,j)
```



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Hybrid programming

Hybrid programming

- MPI is here to stay
  - A lot of HPC applications already written in MPI
  - MPI scales well to tens/hundreds of thousands of nodes

- OpenMP efficient within node
  - Faster communication/interaction
  - More dynamic scheduling capabilities

- Why hybrid?
  - Leverage most appropriate capabilities
  - Might improve load balance
    - If imbalance increases with #mpi processes but not imbalanced at OpenMP level
  - May reduce global communication cost

  - But some negative experiences

Hybrid programming MPI + OpenMP/OmpSs

- Typical MPI+OpenMP practice
  - OpenMP for computation phases
    - Fork - join
  - Serialized communication phases
  - Amdahl´s law for hybrid programming

- MPI + OmpSs/OpenMP potential
  - Propagates asynchronous features to global program behavior
  - Potential for automatic overlap
    - Communication and computation
    - Communication and communication
    - Computation and computation

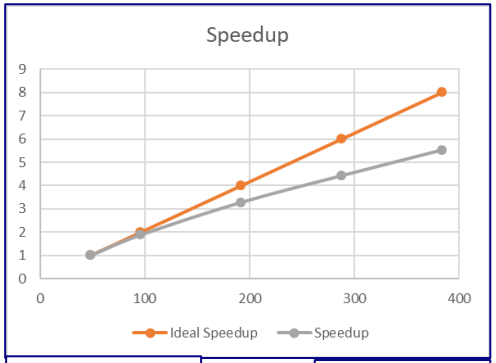
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Don’t mask your symptoms

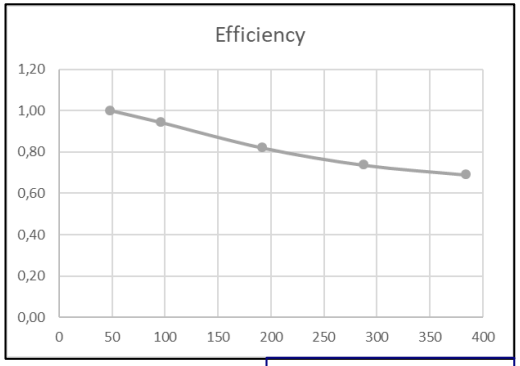
Understand fundamental effects

- Who to blame ?
  - Causes / Effects ?
  - Measurements
    - Too much aggregation ?
    - Too much detail ?



T(48)=11.43 s

$$Sup = \frac{T_{ref}}{T_P}$$

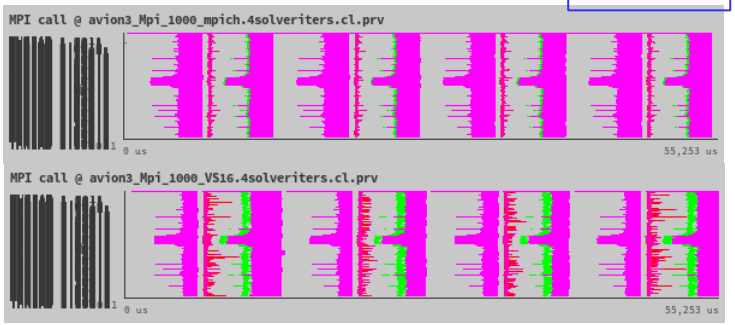


$$Eff = \frac{Sup}{P / ref}$$

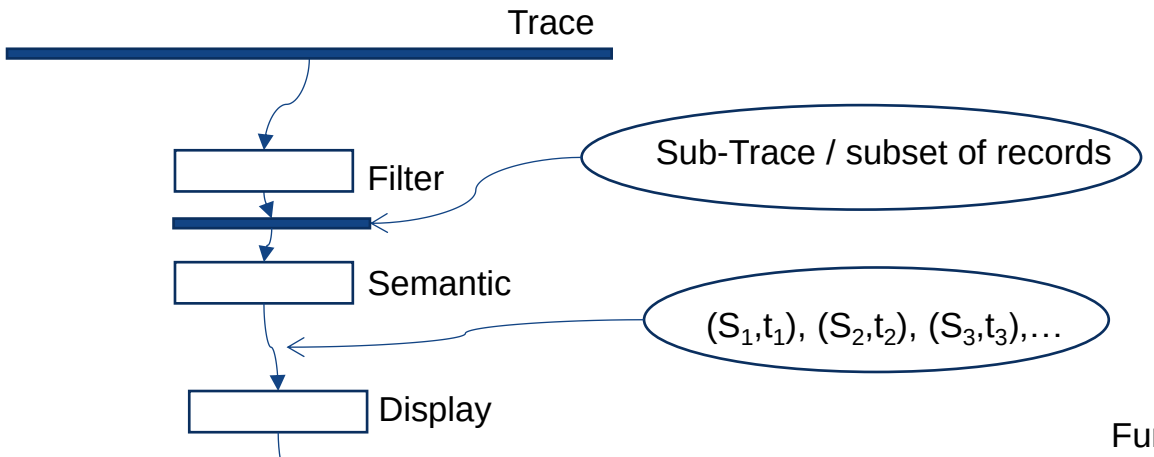
- Understand real issues
  - Abstract model
  - Detail

- Basis to
  - Fix
  - Counteract ← Hybridizing ?

1k procs



Paraver mathematical foundation



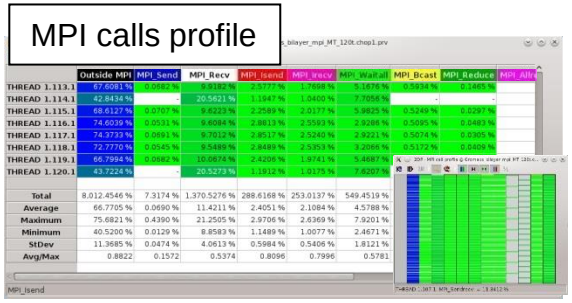
$$s(t) = S_i, t \in [t_i, t_{i+1}), i \in \mathbb{N}$$

Function of time

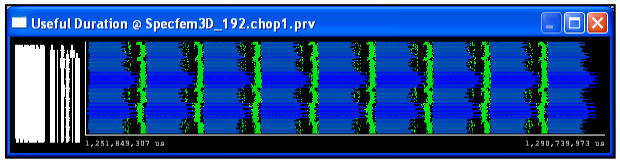
Series of values

MPI calls

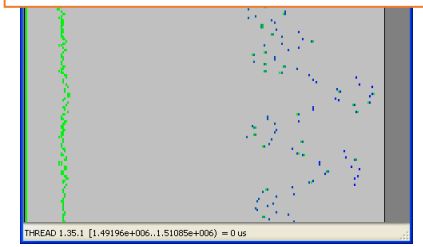
MPI calls profile



Useful Duration

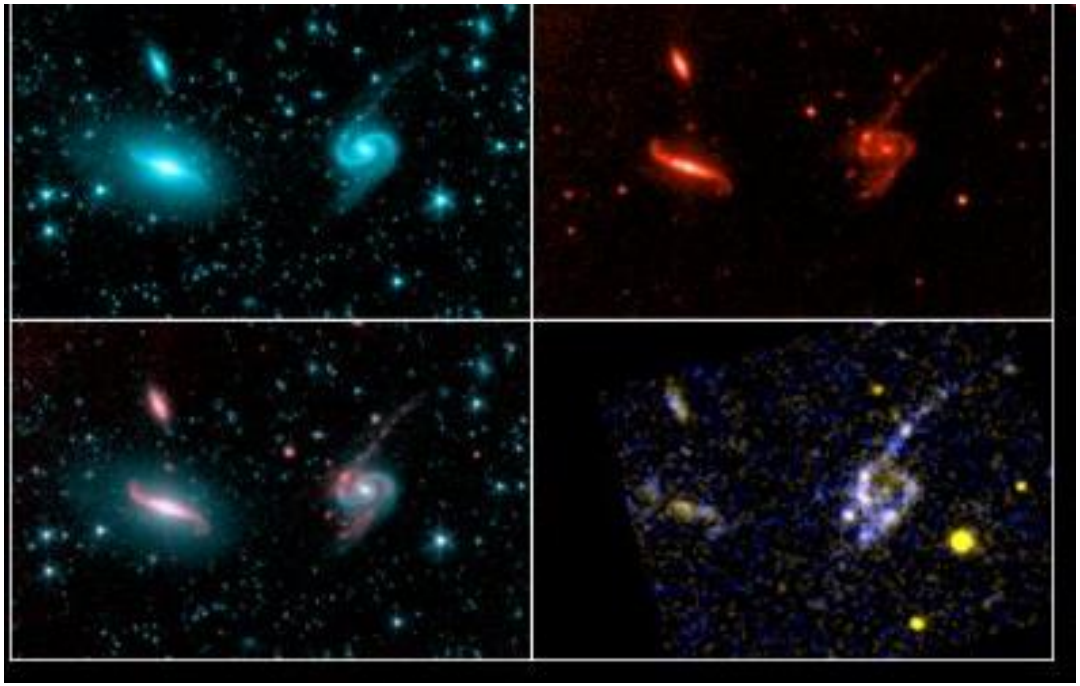


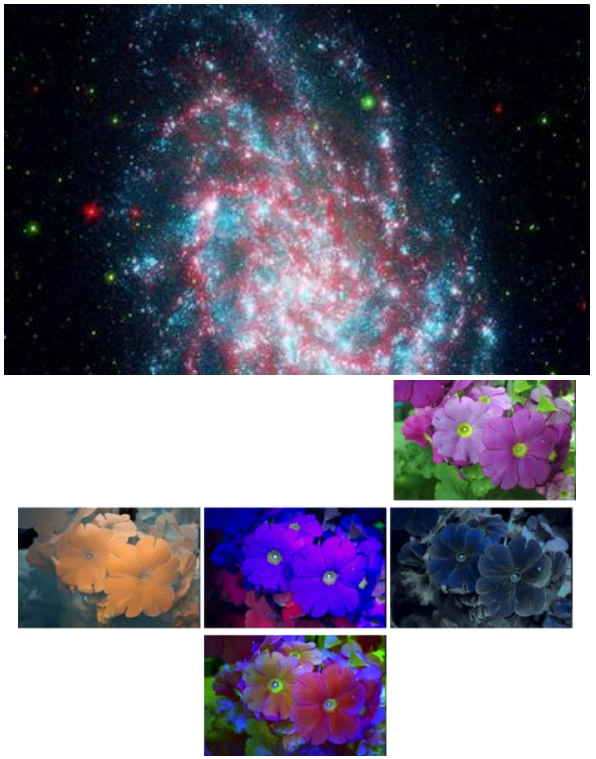
Histogram Useful Duration



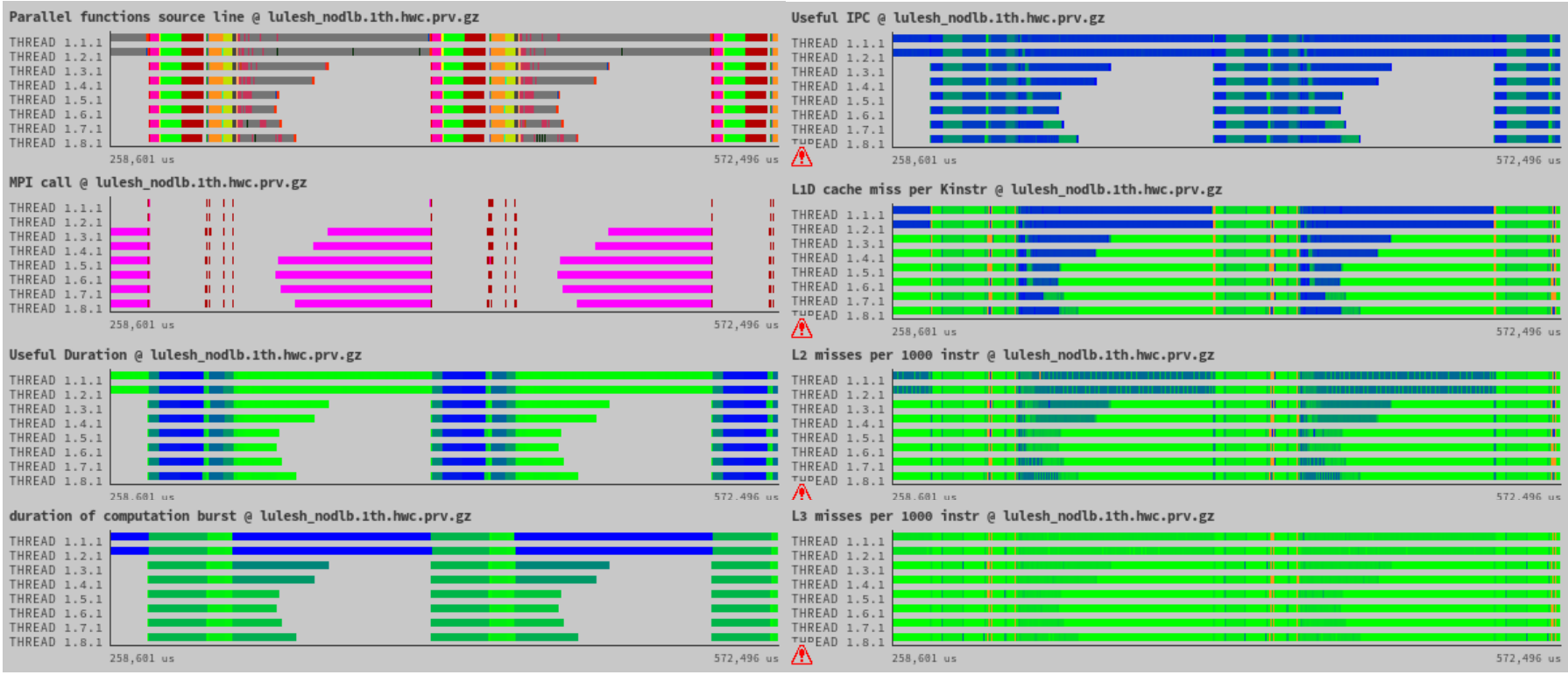
Multispectral imaging

- Different looks at one reality
  - Different spectral bands (light sources and filters)
- Highlight different aspects
  - Can combine into false colored but highly informative images



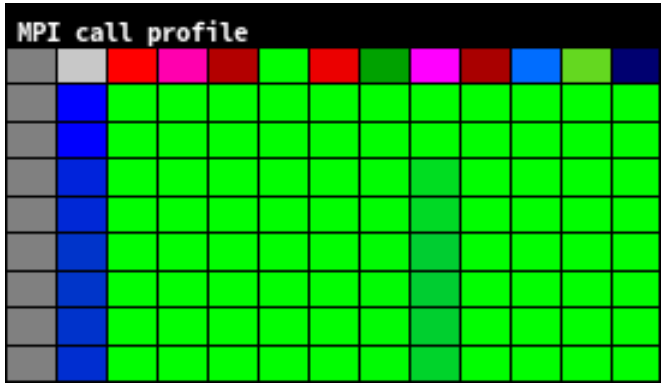


Multispectral timelines

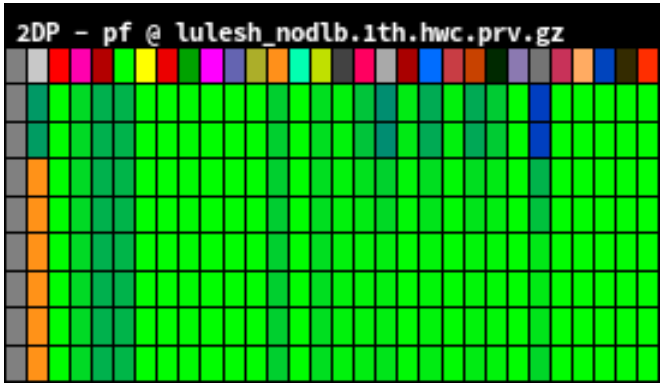


Multispectral statistics

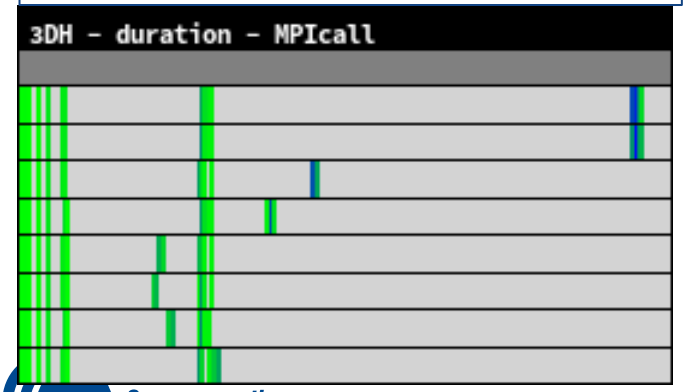
Profile MPI calls



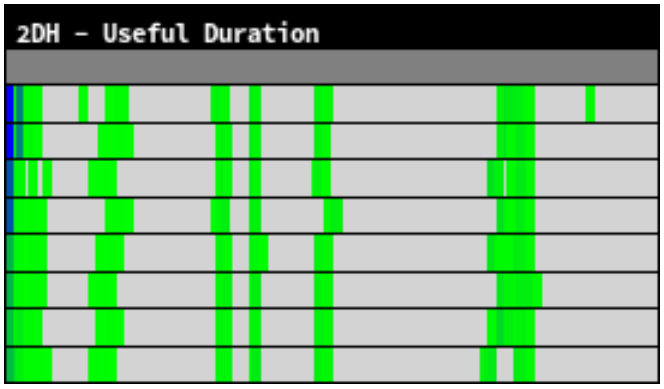
Profile parallel functions



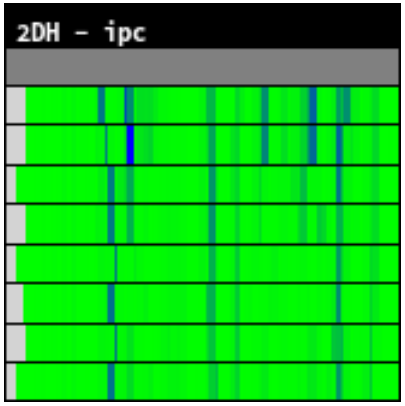
Histogram duration between MPI calls



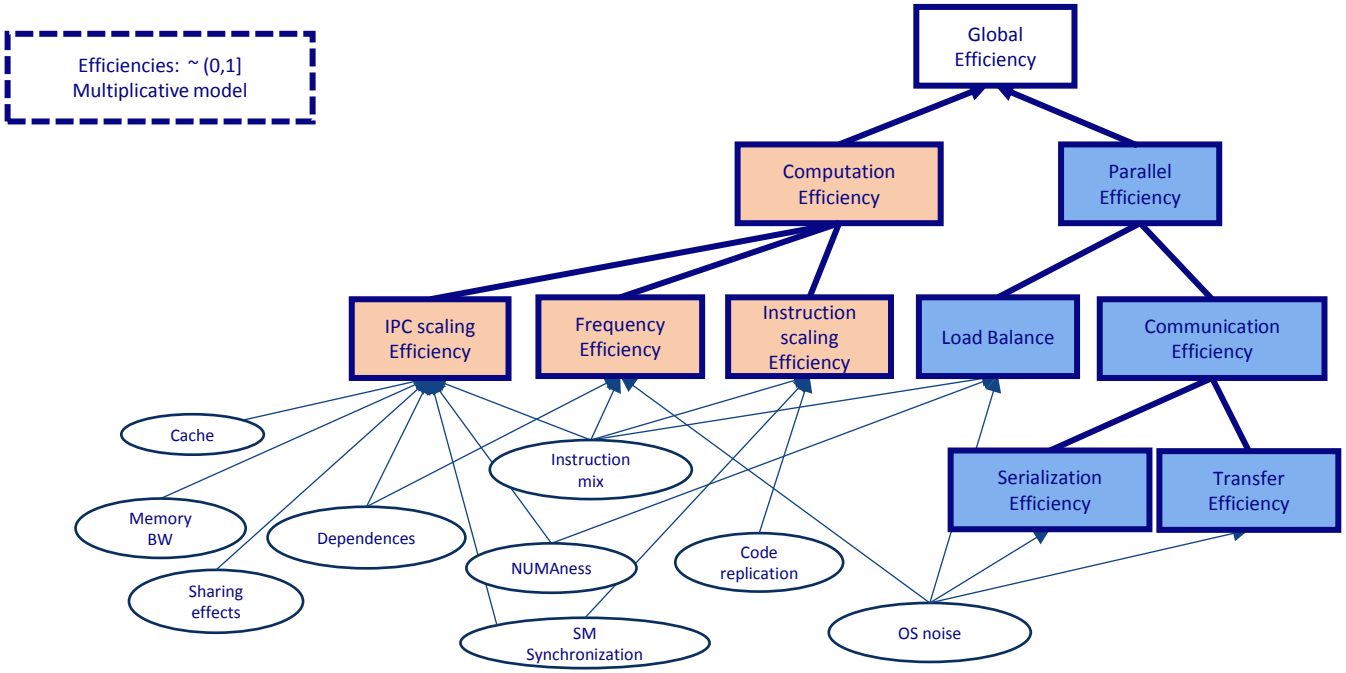
Histogram useful duration



Histogram useful IPC



Hierarchical Performance Model



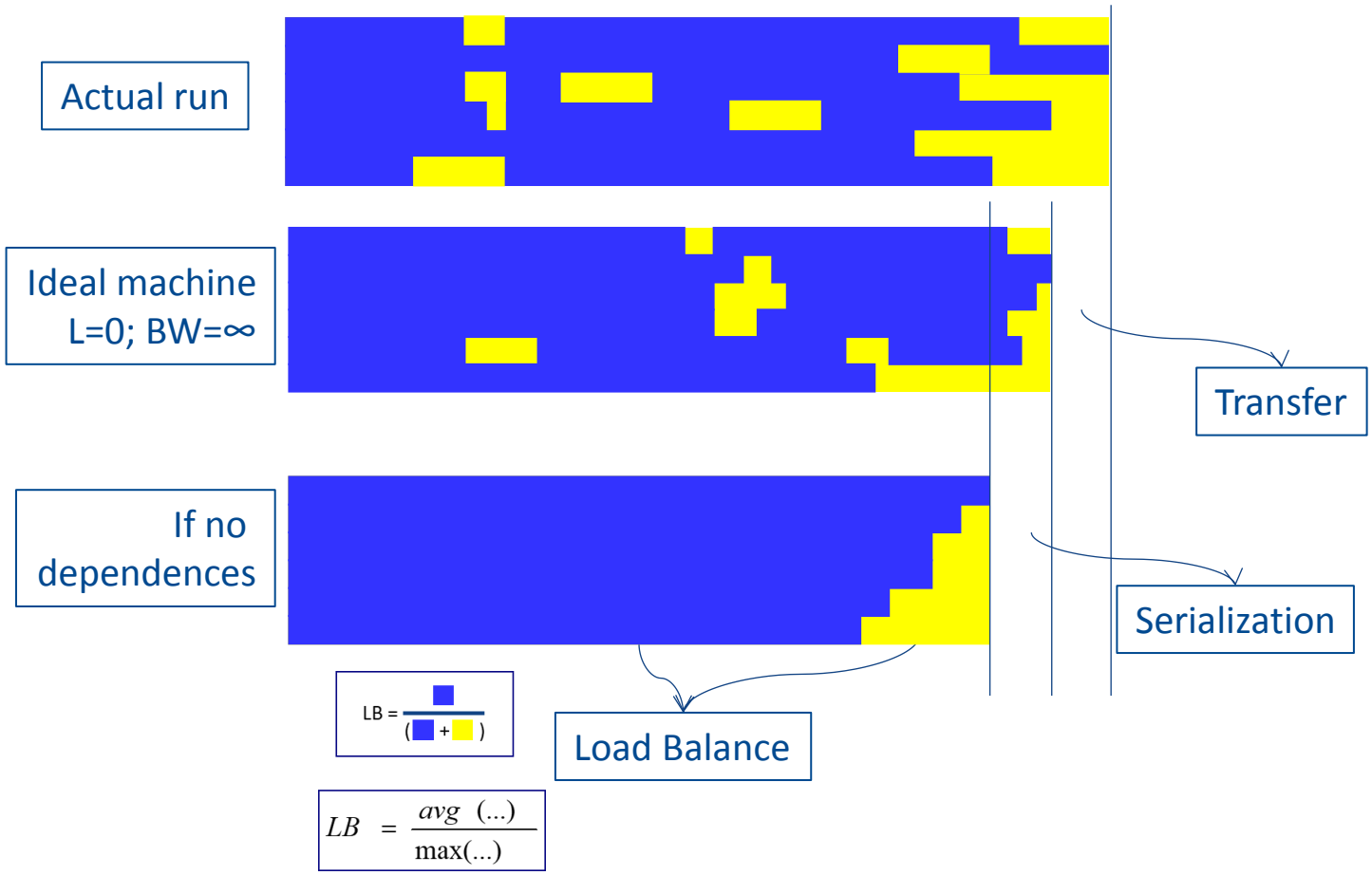
$$CompEff = Ieff * IPCeff * Feff$$

$$\eta_{\parallel} = LB * Ser * Trf$$

$CommEff$

M. Casas et al, "Automatic analysis of speedup of MPI applications". ICS 2008.

Parallel efficiency model



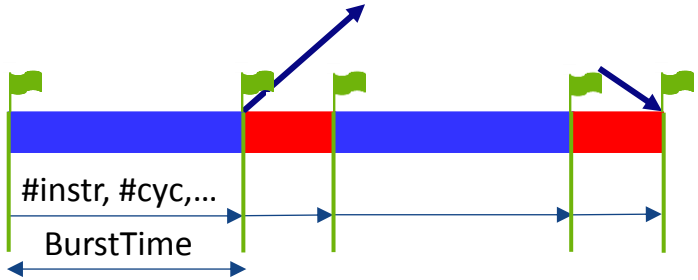
$$LB = \frac{\frac{1}{P}\sum_{i=1}^{P} c_i}{\max(c_i)}$$

$$Ser = \frac{\max(c_i)}{T_{ideal}}$$

$$Trf = \frac{T_{ideal}}{T}$$

Characterizing sequential performance

- Computation efficiency model
  - Performance scaling factor over reference case
    - 0 .. 1 .. X
  - Multiplicative model
- Efficiency factors
  - Instruction scaling efficiency
    - Total amount of work. Code replication?
  - IPC scaling efficiency
    - How fast are instructions executed by architecture
  - Frequency efficiency
    - Frequency changes with load



$$InstrEff = \frac{\#instr_{ref}}{\#instr_{P}}$$

Reference core count

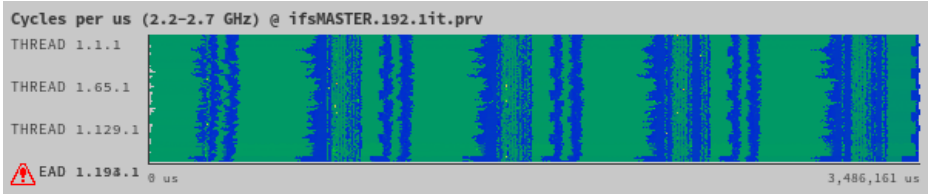
$$IPCEff = \frac{IPC_{P}}{IPC_{ref}}$$

$$Feff = \frac{F_{P}}{F_{ref}}$$

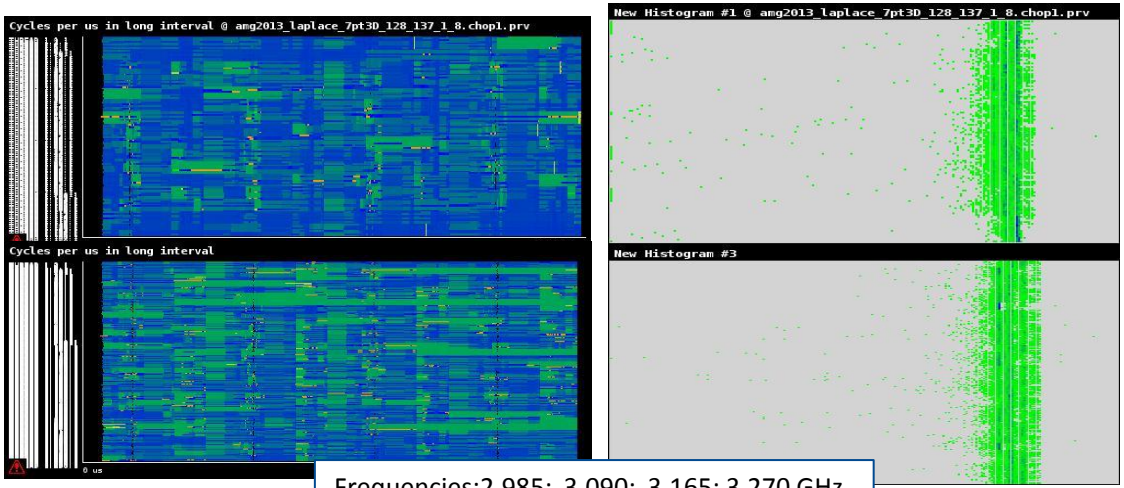
In user level code "Useful"

A comment on frequency

- Good old times where frequency was known are gone
  - Turbo
  - DVFS
  - Power capping, governors
  - Device variability
- PAPI counters virtualized
  - OS activity, noise, yields



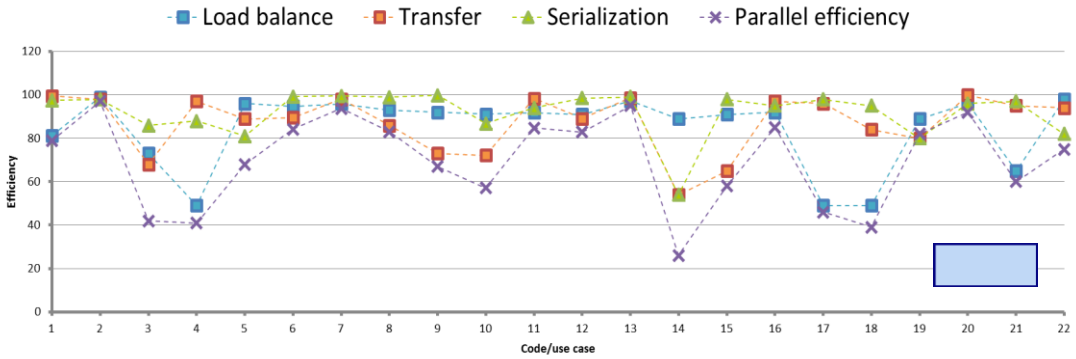
Towards unpredictable core performance

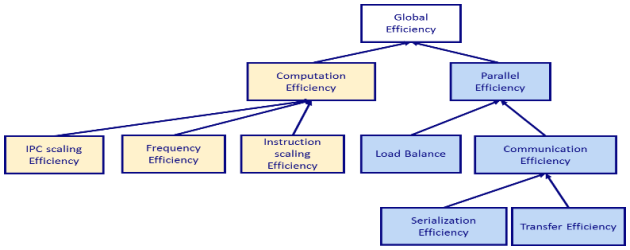


Frequencies:2.985; 3.090; 3.165; 3.270 GHz

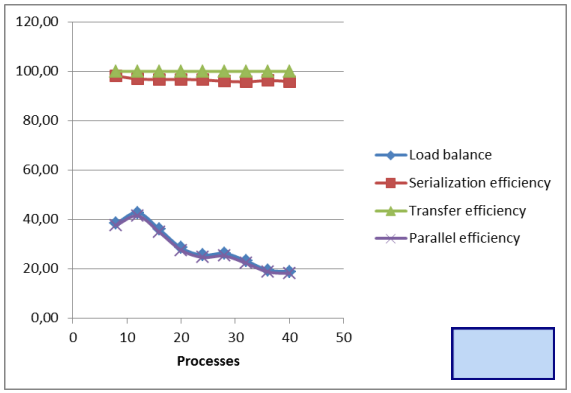
Behavior awareness

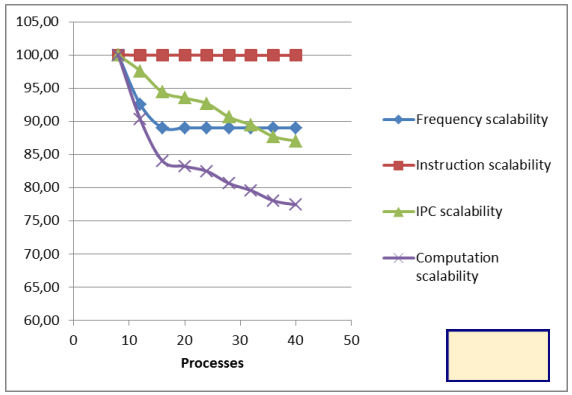
- A common language about fundamental issues





- Evolution of bottlenecks



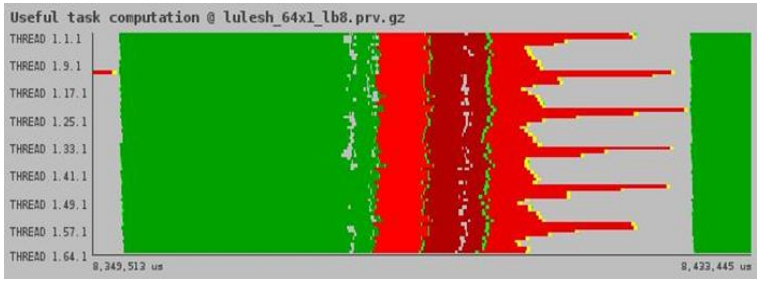


https://pop-coe.eu

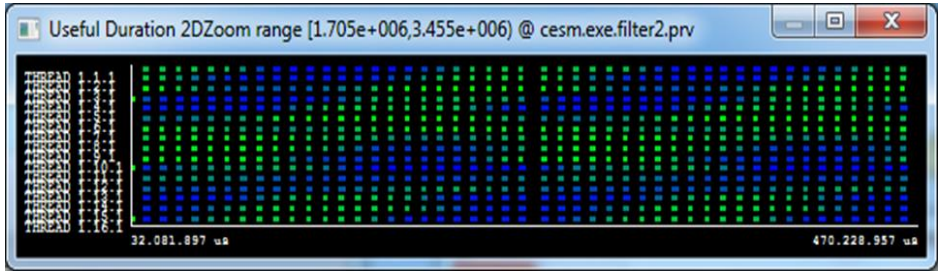
Imbalances

- Structural characteristics
  - Global / Local
  - Repetitive / Migrating / Sporadic pattern
- Causes
  - Computational
  - Non parallelized regions
  - IPC
    - Locality
    - Contention
  - Communication
  - Noise
  - Runtime implementation
  - Memory layout

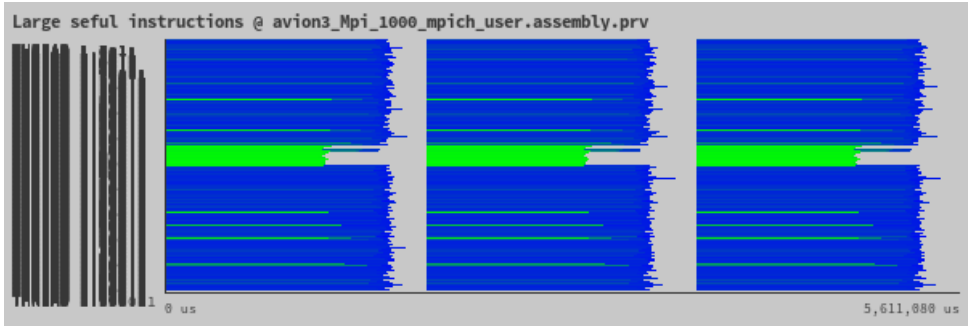
Lulesh



CESM

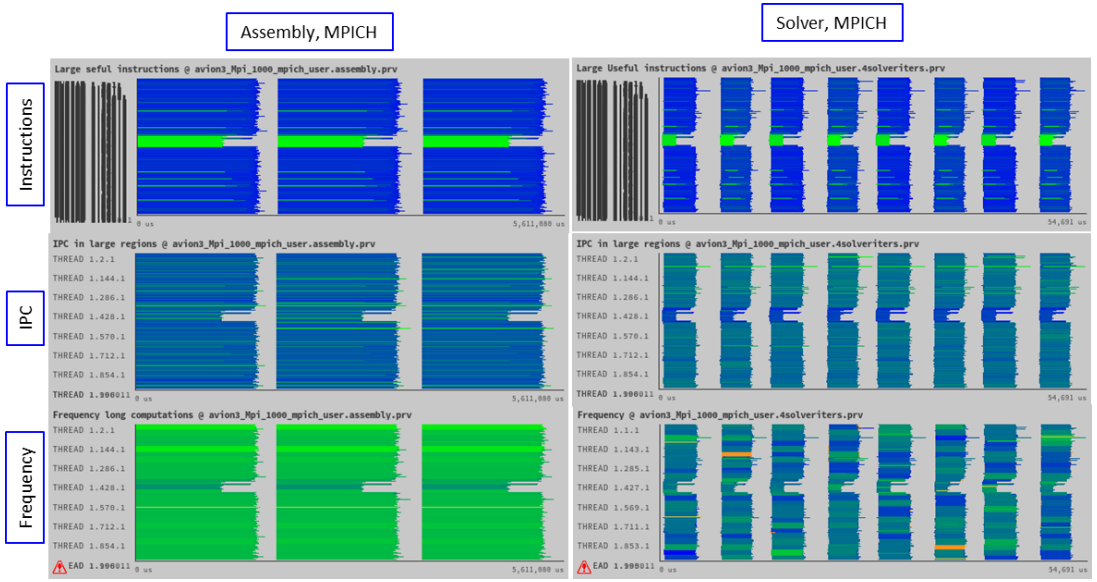


Alya

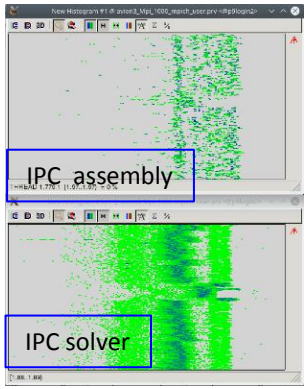


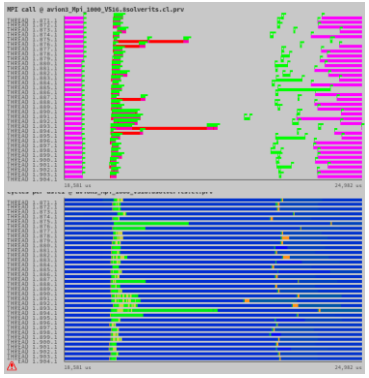
Imbalances

- Often not addressable by static methods

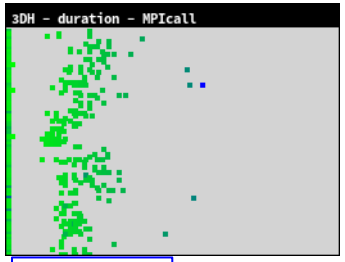


Alya – Full Plane @ 1K cores



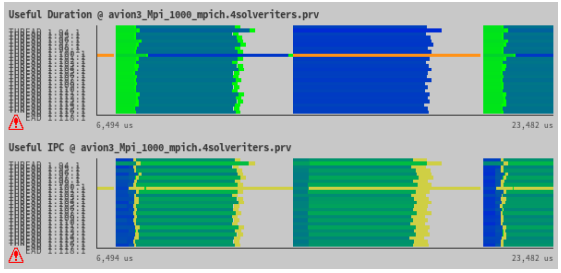


Isend duration @solver, OpenMPI



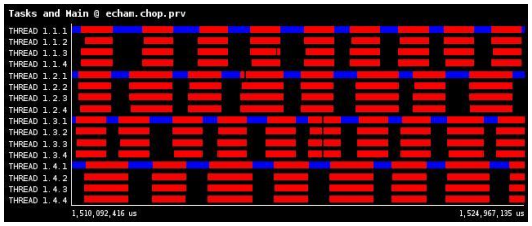
700-900 us

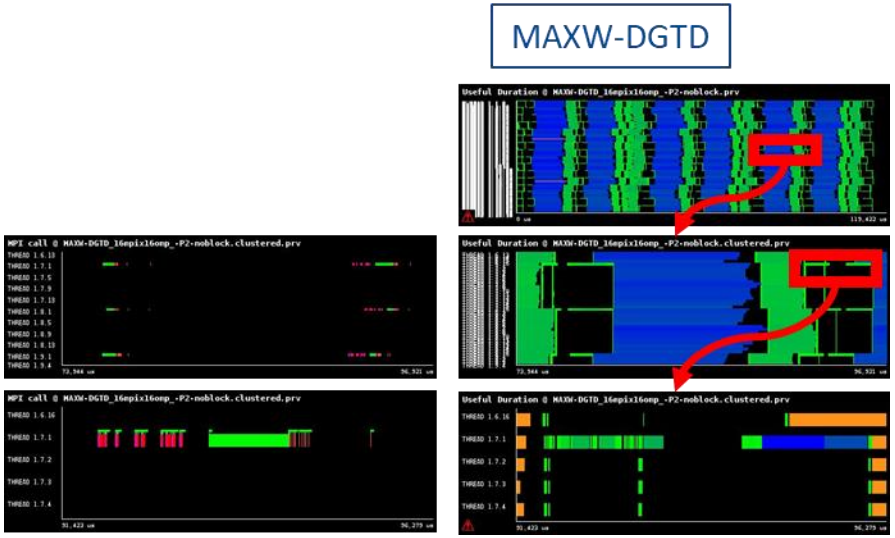
IPC imbalance @solver, MPICH



Amdahl's Law for hybrid programming

- Hybrid Amdahl's law
  - A fairly "bad message" for programmers
- "Reasons"
  - Bottom up "bottleneck" incremental approach
  - "Lazy" programmer
  - Coupled codes
    - Configuration challenge
    - Multiple developers
  - MPI serialization





Don’t mask your symptoms

- Analyze the pure MPI/single thread behavior
  - Structure:
    - Overall understanding
    - Focus of Analysis
  - Scalability
  - Efficiency model
  - Actual causes
    - Load balance: Useful duration, useful instructions, IPC, Frequency (noise), unparallelized regions, …
    - Communication: compare actual run to ideal run
      - Serialization: dependence chains in ideal run
      - Transfer: communication times that vanish from actual to ideal run, message sizes,…
- An then plan counter measures

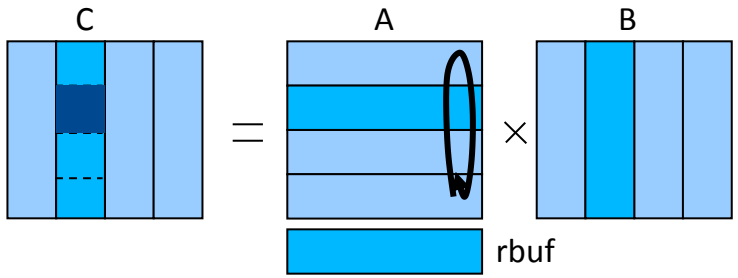
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

My first hybrid program

My first hybrid program

```
…
for( it=0; it<nodes; it++ ) {


   mxm( m, n, a, B, (double (*)[n])&C[i][0]);

   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;

}

…
```

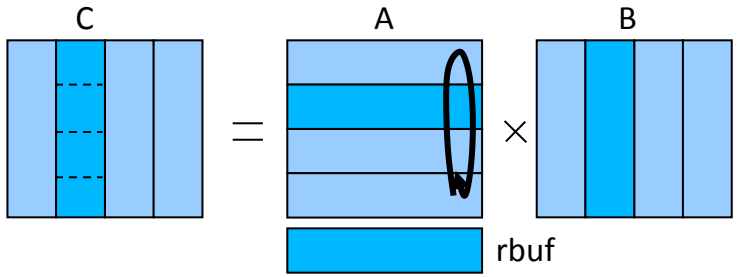



```
void mxm ( int m, int n, double *a,
           double *b, double *c ) {
   …
   for (i=0; i<n; i++) {

      dgemm(…, a, b, c);
      c+=n; a+=m;
   }

}
```

```
void callSendRecv(int m, int n,
         double (*a)[m], int down,
         double (*rbuf)[m], int up)
{
   …
   MPI_Sendrecv( a, …, rbuf, …);
}
```

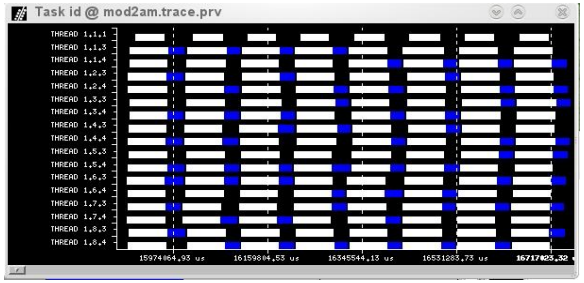
My first hybrid program

```
…
for( it=0; it<nodes; it++ ) {


   mxm( m, n, a, B, (double (*)[n])&C[i][0]);

   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;

}

…
```



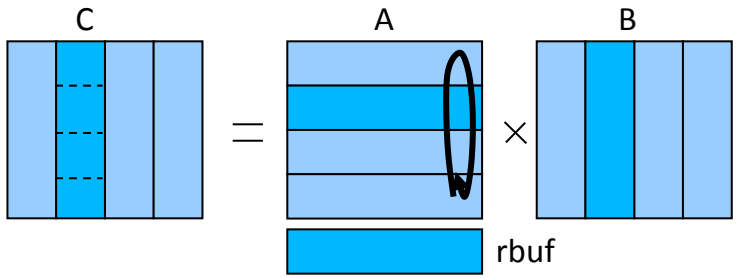

```
void mxm ( int m, int n, double *a,
           double *b, double *c ) {
   …
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(…, a, b, c);
      c+=n; a+=m;
   }
   #pragma omp taskwait
}
```

```
void callSendRecv(int m, int n,
         double (*a)[m], int down,
         double (*rbuf)[m], int up)
{
   …
   MPI_Sendrecv( a, …, rbuf, …);
}
```



My first hybrid program

```
…
for( it=0; it<nodes; it++ ) {


   mxm( m, n, a, B, (double (*)[n])&C[i][0]);

   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;
 #pragma omp taskwait
}

…
```



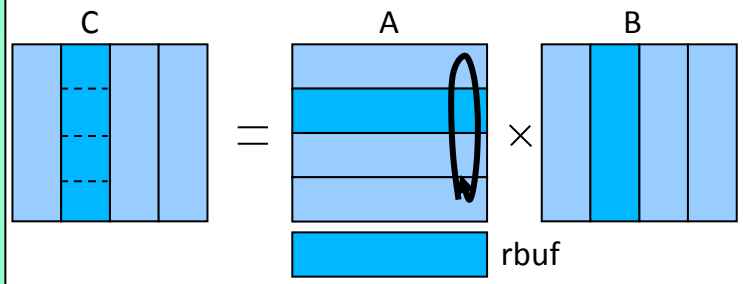

```
void mxm ( int m, int n, double *a,
           double *b, double *c ) {
   …
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(…, a, b, c);
      c+=n; a+=m;
   }

}
```

```
void callSendRecv(int m, int n,
         double (*a)[m], int down,
         double (*rbuf)[m], int up)
{
   …
   MPI_Sendrecv( a, …, rbuf, …);
}
```

- Overlap computation in tasks and communication in master

My first hybrid program

```
…
for( it=0; it<nodes; it++ ) {


   mxm( m, n, a, B, (double (*)[n])&C[i][0]);
   #pragma omp task in([n][m]a) out([n][m]rbuf)
   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;
 #pragma omp taskwait
}

…
```



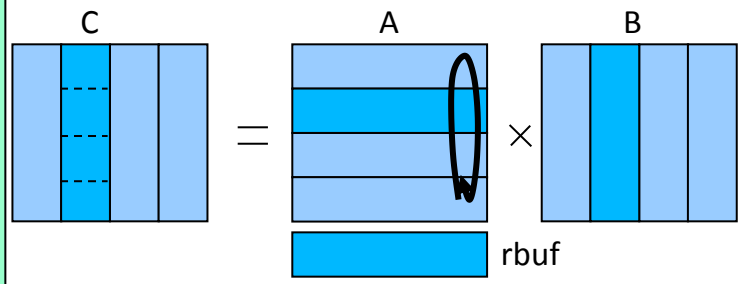

```
void mxm ( int m, int n, double *a,
           double *b, double *c ) {
   …
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(…, a, b, c);
      c+=n; a+=m;
   }

}
```

```
void callSendRecv(int m, int n,
         double (*a)[m], int down,
         double (*rbuf)[m], int up)
{
   …
   MPI_Sendrecv( a, …, rbuf, …);
}
```

- Overlap between computation and communication in tasks

My first hybrid program

```
…
for( it=0; it<nodes; it++ ) {
   #pragma omp task in([n][m]a, B) \
                    inout(C[i;n][0;n])
   mxm( m, n, a, B, (double (*)[n])&C[i][0]);
   #pragma omp task in([n][m]a) out([n][m]rbuf)
   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;
 #pragma omp taskwait
}

…
```



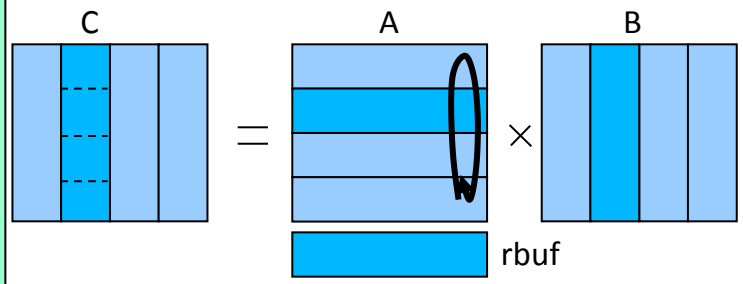

```
void mxm ( int m, int n, double *a,
           double *b, double *c ) {
   …
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(…, a, b, c);
      c+=n; a+=m;
   }
   #pragma omp taskwait
}
```

```
void callSendRecv(int m, int n,
         double (*a)[m], int down,
         double (*rbuf)[m], int up)
{
   …
   MPI_Sendrecv( a, …, rbuf, …);
}
```

- Nested.
- Overlap between computation and communication in tasks

My first hybrid program

```
…
for( it=0; it<nodes; it++ ) {
   #pragma omp task in([n][m]a, B) \
                    inout(C[i;n][0;n])
   mxm( m, n, a, B, (double (*)[n])&C[i][0]);
   #pragma omp task in([n][m]a) out([n][m]rbuf)
   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;

}
#pragma omp taskwait
…
```



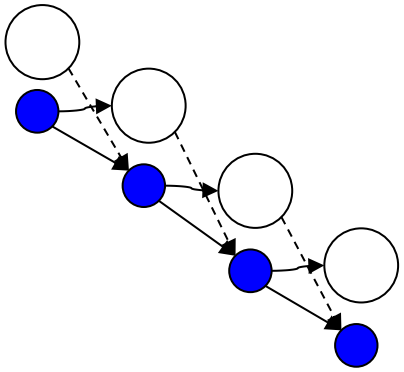

```
void mxm ( int m, int n, double *a,
           double *b, double *c ) {
   …
   for (i=0; i<n; i++) {
      #pragma omp task
      dgemm(…, a, b, c);
      c+=n; a+=m;
   }
   #pragma omp taskwait
}
```

```
void callSendRecv(int m, int n,
         double (*a)[m], int down,
         double (*rbuf)[m], int up)
{
   …
   MPI_Sendrecv( a, …, rbuf, …);
}
```

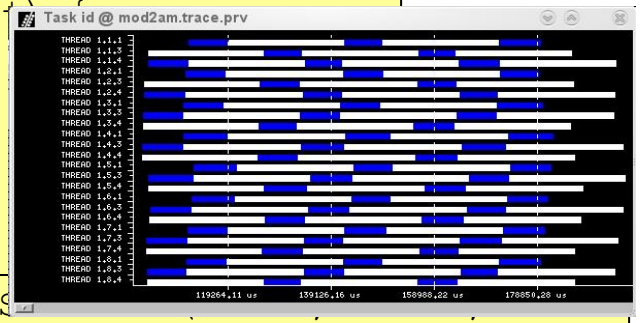
- Nested.
- Can start computation of next block of C as soon as communication terminates

My first hybrid program

```
…
for( it=0; it<nodes; it++ ) {
   #pragma omp task in([n][m]a, B) \
                    inout(C[i;n][0;n])
   mxm( m, n, a, B, (double (*)[n])&C[i][0]);
   #pragma omp task in([n][m]a) out([n][m]rbuf)
   callSendRecv(m, n, a, down, rbuf, up);

   //next C block circular
   i = (i+n)%m;
   //swap pointers
   ptmp=a; a=rbuf; rbuf=ptmp;

}
#pragma omp taskwait
…
```



```
void mxm ( int m, int n, double *a,
           double *b, double *c ) {
   …
   for (i=0; i<n; i+
      dgemm(…, a, b,
      c+=n; a+=m;
   }

}
```



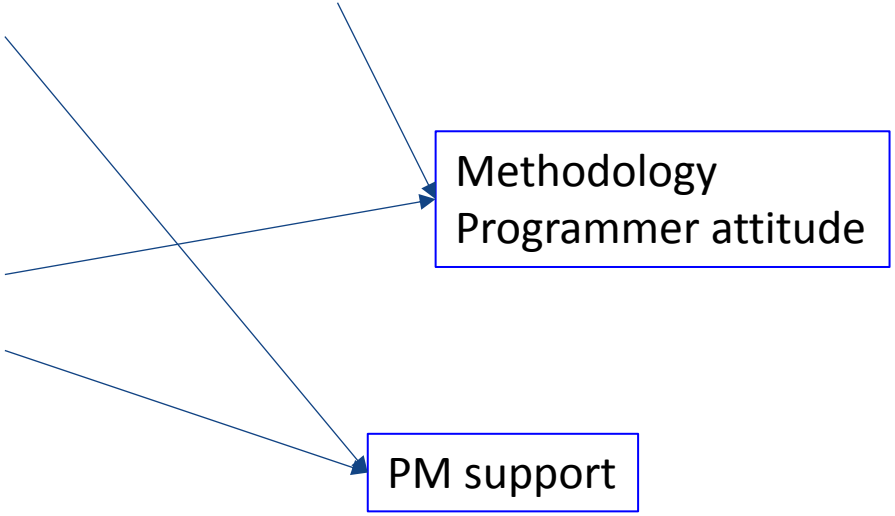

```
void calls
          double (*a)[m], int down,
          double (*rbuf)[m], int up)
{
   …
   MPI_Sendrecv( a, …, rbuf, …);
}
```

- Can obtain parallelism without parallelizing fine grain the computation

Thoughts

- Flexibility !!!
  - Small syntax changes significant behavior / execution orders
  - Potential surprise (unexpected possibilities)

- Some “issues”
  - Bottom up vs. top down
  - Nesting & composability



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

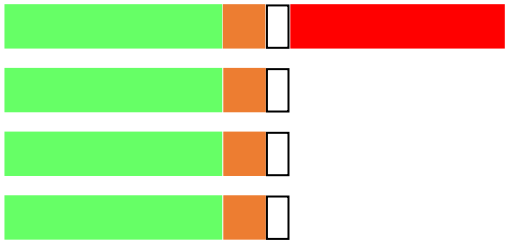
EXCELENCIA SEVERO OCHOA

Taskify Computation

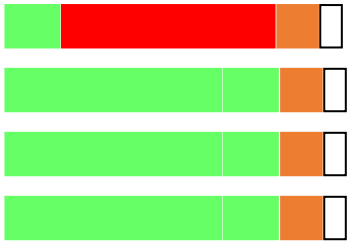
Taskigy computation: a chance for lazy programmers

Four loops/routines
Sequential program order

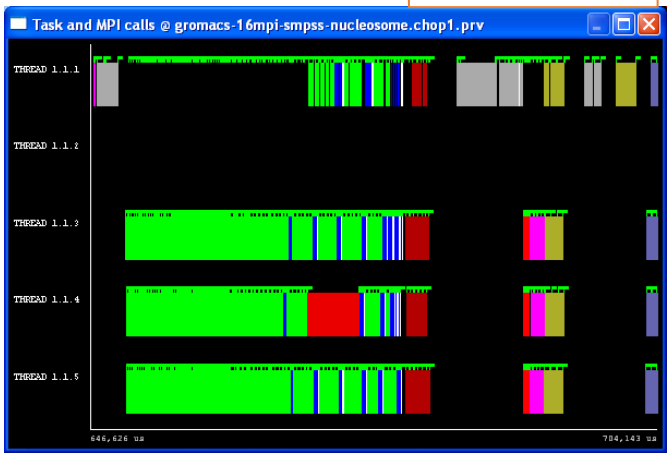
OpenMP
not parallelizing one loop



SMPSs
not parallelizing one loop



GROMACS



Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Taskify Communications

Interaction between MPI and Task models

- Communication can be taskified or not.

- If not taskified:
  - still potential to overlap communication performed by main thread with previously generated tasks
  - Careful synchronization
  - Stall control flow lookahead

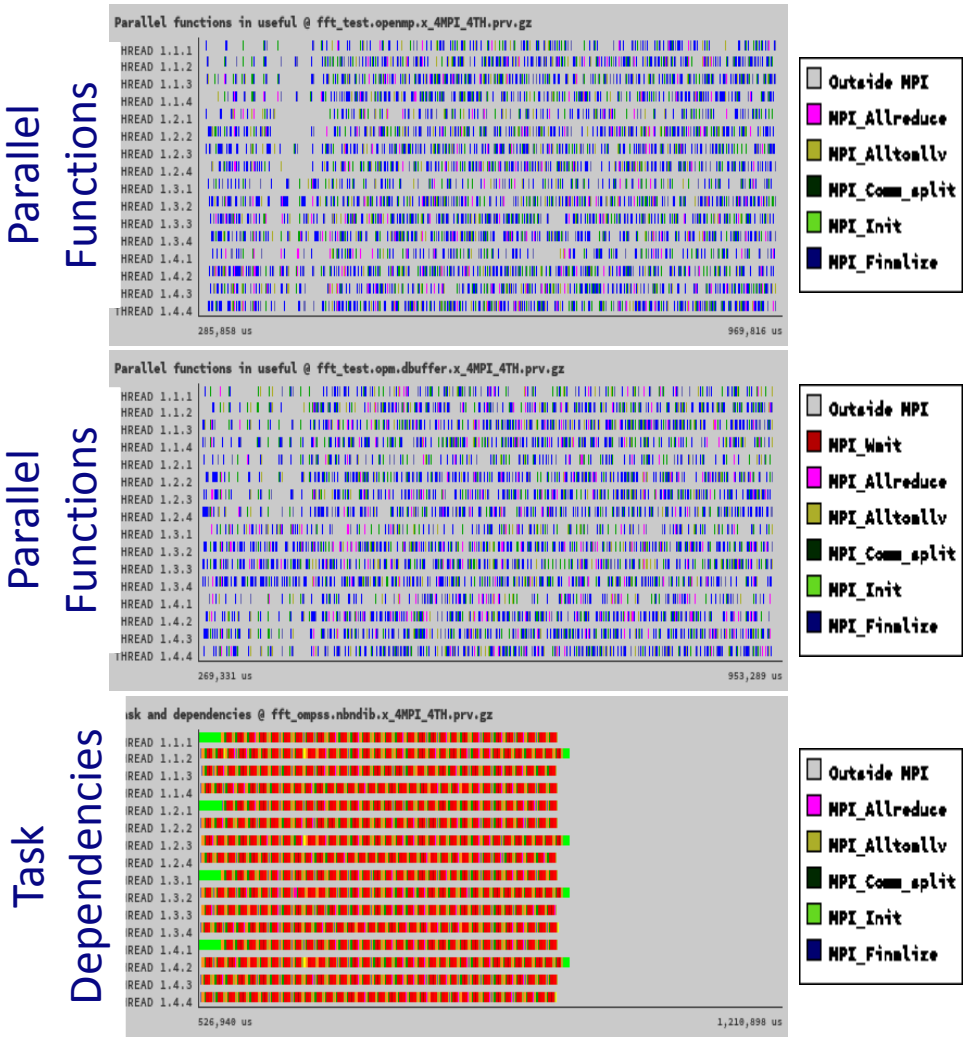
- If taskified:
  - Lookahead: potential for overlap/out of order execution
    - Computation - communication
    - Communication - communication
  - overlap instantiation overhead → dependence

To consider when taskifying comms.

- Possibly several concurrent MPI calls → thread safe MPI
  - Might not be available or really concurrent in some installations
- MPI too serial
  - Internal MPI state is also shared state
  - Matching semantics
- MPI tasks waste cores if communication partner delays or long transfer times
  - Less problem as more and more cores per node become available
- Reordering of MPI calls + blocking calls + limited number of cores → potential to introduce deadlock
  - → need to control order of communication tasks

MPI too serial: internal state

- MPI + OpenMP
- Non blocing MPI + OpenMP
- MPI + OmpSs
  - Top down
    - Overlap communications
    - Serial FFTs
  - Replicate communicators

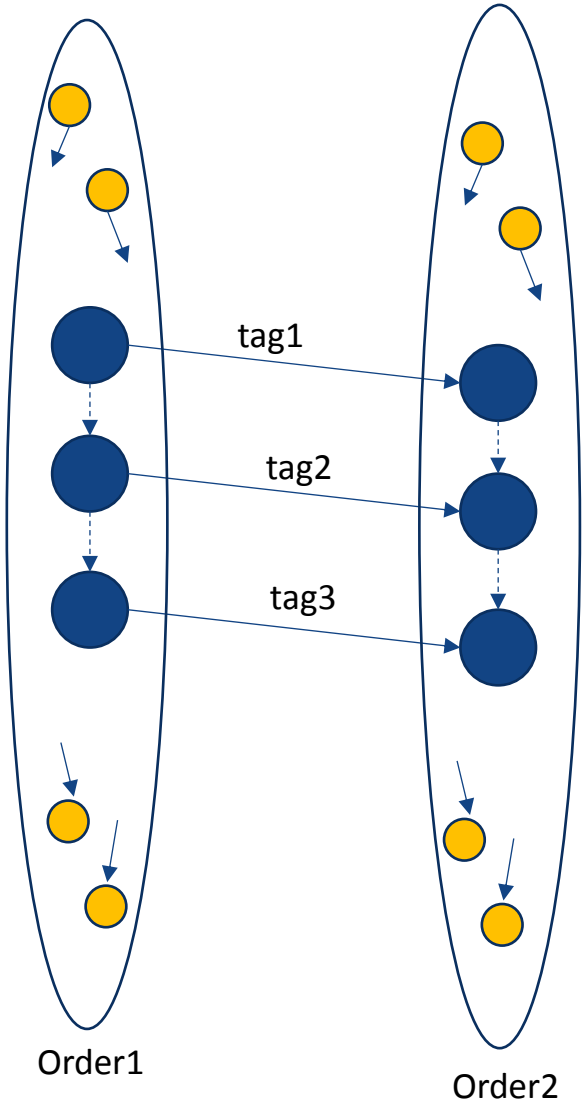


FFTlib - Quantum Espreso mini app

MPI too serial: matching semantics

- Have been widely relied on
- Resulting in very “serial” code

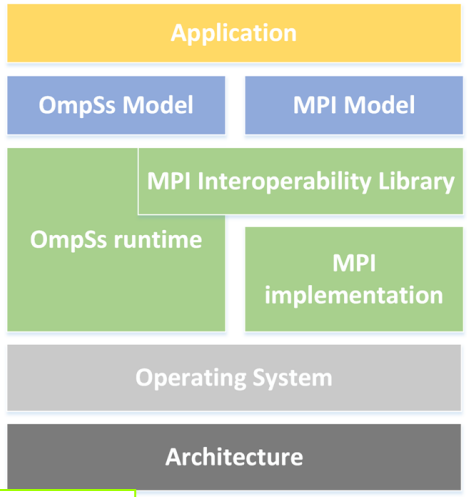
- May require
  - to enforce order among MPI tasks
    - Dependences through sentinel
  - More precisely specify matching



Suggestion:
Leave order to OpenMP
Do not mask your sympthoms

Deadlock potential when taskifying communications

- Approaches to handle
  - Algorithmic structure may not expose loop and thus be deadlock free
  - Enforce in order execution of communicating tasks
    - Dependences
    - Inout(sentinel) on all communication tasks
    - Serialization of communication can have an impact on performance
  - Virtualize communication resource. Allow infinite communication tasks
    - MPI-OmpSs interoperability library (TAMPI)
      - Blocking MPI calls → nonblocking
      - Block task in Nanos runtime
      - Poll completion
      - Reactivate task when communication completes
    - Also addresses issues of wasted cores
      - Interaction with task granularity



V. Marjanovic et al, "Overlapping Communication and Computation by using a Hybrid MPI/SMPSs Approach" ICS 2010

K. Sala et al, "Improving the Interoperability between Task-Based and Distributed Programming Models". EuroMPI 2018

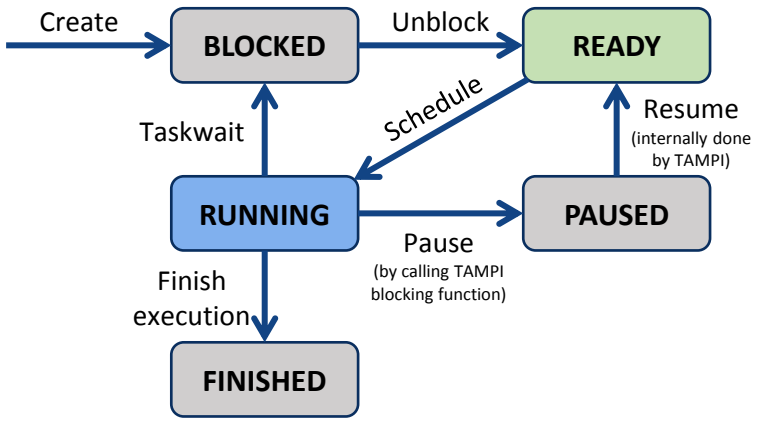
Task-Aware MPI (TAMPI)

- Library to virtualize cores for unlimited number of blocked tasks in MPI
-
- Two modes
  - **Blocking** Mode:
    - Support for **blocking MPI calls** inside tasks (*MPI_Recv*, *MPI_Bcast*...)
    - Supported just by **OmpSs-2** ☹
  - **Non-Blocking** Mode:
    - Support for **non-blocking MPI calls** inside tasks (*MPI_Issend*, *MPI_Igatherv*...)
    - Supported by both **OpenMP** and **OmpSs-2** ☺

K. Sala et al, "Improving the Interoperability between Task-Based and Distributed Programming Models". EuroMPI 2018

Blocking Mode (OmpSs-2)

- Support for blocking MPI calls inside tasks (MPI_Recv, MPI_Bcast...)
  - Tasks will not use compute resources (cores) while in the blocking call
- Virtualizes the execution resources (e.g., hardware threads)
  - MPI calls are intercepted
  - The calling tasks are “paused”, the core is handed over to execute another ready task
  - When the call completes, the task is resumed (made ready by the runtime)
- Implementation requires coordination between MPI and shared memory runtime



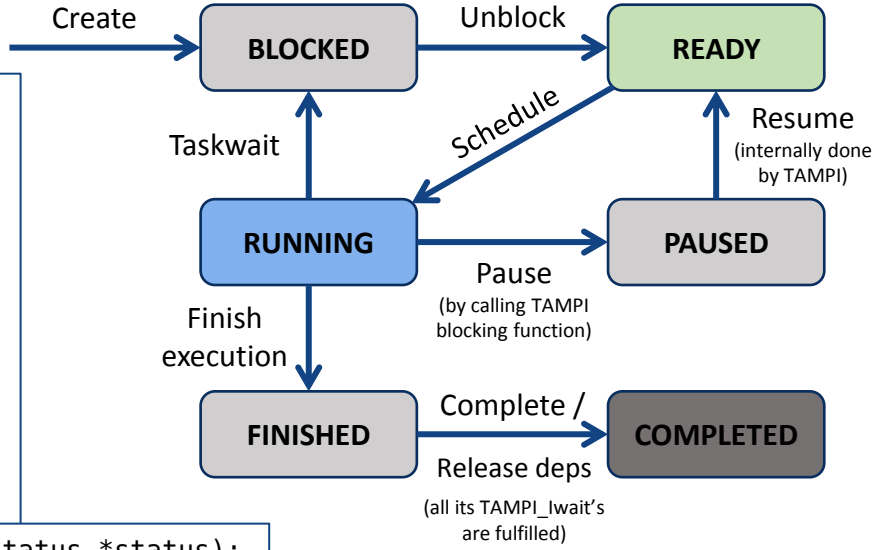

```
#include <TAMPI.h>
// ...
#pragma oss task in(senddata[i]) out(recvdata[i])
{
  MPI_Send(&senddata[i], 1, MPI_INT, 1, tag,
                      MPI_COMM_WORLD);
  // The send() operation has already been completed

  MPI_Recv(&recvdata[i], 1, MPI_INT, 0, tag,
           MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  // The recv() operation has already been completed

  printf("%d", recvdata[i]);
}
```

Non-Blocking Mode (OpenMP & OmpSs-2)

- Support for non-blocking MPI calls inside tasks (MPI_Issend, MPI_Igatherv...)
- Special MPI stub routines for MPI wait calls
  - MPI_wait, MPI_waitall → **TAMPI_Iwait, TAMPI_Iwaitall**
- Semantic implications
  - Task will not be completed (free dependences) till all non blocking calls it issues have actually completed
  - Must specify the data sent or received as task dependences
  - TAMPI Wait calls are outside the task become empty functions

```
#pragma omp task depend(out: recvdata[i], status)
{
  MPI_Request request;
  MPI_Irecv(&recvdata[i], 1, MPI_INT, 0, tag,
            MPI_COMM_WORLD, &request);
  TAMPI_Iwaitall(1, &request, &status);
  // recvdata and status cannot be accessed yet!
}
#pragma omp task depend(in: recvdata[i], status)
{
  check_status(&status);
  process_data(&recvdata[i]);
}
```

```
int TAMPI_Iwait(MPI_Request *request, MPI_Status *status);
int TAMPI_Iwaitall(int count, MPI_Request *requests,
                   MPI_Status *statuses);
```



TAMPI

- MPI_THREAD_MULTIPLE mode in MPI_Init_thread call enables no-blocking mode
- MPI_TASK_MULTIPLE mode in MPI_Init_thread call enables blocking and non-blocking mode
- Same application code for both modes

Wrapper Function	TAMPI Disabled	MPI_THREAD_MULTIPLE	MPI_TASK_MULTIPLE
TAMPI_Irecv	MPI_Irecv	MPI_Irecv + **TAMPI_Iwait**	MPI_Irecv + **TAMPI_Iwait**
TAMPI_Isend	MPI_Isend	MPI_Isend + **TAMPI_Iwait**	MPI_Isend + **TAMPI_Iwait**
TAMPI_Ibcast	MPI_Ibcast	MPI_Ibcast + **TAMPI_Iwait**	MPI_Ibcast + **TAMPI_Iwait**
…	…		…
TAMPI_Wait	MPI_Wait	-	-
TAMPI_Waitall	MPI_Waitall	-	-
MPI_Send	MPI_Send	MPI_send (busy wait. May deadlock)	MPI_send (virtualized core)

IFSKernel example

A. Original Pure MPI

```
include “mpi.h”
! ...
nreqs=0
tag=1
do proc=1,nprocs
  if (sendoff(proc) > 0) then
    nreqs=nreqs+1

    call MPI_Isend  (senddata(:,proc), sendoff(proc),
            MPI_REAL8, proc-1, tag, MPI_COMM_WORLD,
            reqs(nreqs), ierr)

  endif
  if (recvoff(proc) > 0) then
    nreqs=nreqs+1

    call MPI_Irecv  (recvdata(:,proc), recvoff(proc),
            MPI_REAL8, proc-1, tag, MPI_COMM_WORLD,
            reqs(nreqs), ierr)

  endif
enddo
call MPI_Waitall  (nreqs, reqs(:), statuses(:), ierr)

do proc=1,nprocs
  if (recvoff(proc) > 0) then

    call process_data(recvdata(:,proc), recvoff(proc))

  endif
enddo

! ...
```

mpif90 test.f90

IFSKernel example

B. OmpSs tasks + TAMPI

```
#include “TAMPIf.h”
include “mpi.h”
! ...
nreqs=0
tag=1
do proc=1,nprocs
  if (sendoff(proc) > 0) then
    nreqs=nreqs+1
    !$OSS TASK ... IN(senddata(:,proc))
    call TAMPI_Isend(senddata(:,proc), sendoff(proc),
            MPI_REAL8, proc-1, tag, MPI_COMM_WORLD,
            reqs(nreqs), ierr)
    !$OSS END TASK
  endif
  if (recvoff(proc) > 0) then
    nreqs=nreqs+1
    !$OSS TASK ... OUT(recvdata(:,proc), statuses(nreqs))
    call TAMPI_Irecv(recvdata(:,proc), recvoff(proc),
            MPI_REAL8, proc-1, tag, MPI_COMM_WORLD,
            reqs(nreqs), statuses(nreqs), ierr)
    !$OSS END TASK
  endif
enddo
call TAMPI_Waitall(nreqs, reqs(:), statuses(:), ierr)

do proc=1,nprocs
  if (recvoff(proc) > 0) then
    !$OSS TASK ... IN(recvdata(:,proc))
    call process_data(recvdata(:,proc), recvoff(proc))
    !$OSS END TASK
  endif
enddo
! ...
```

mpif90 -fc=mfc –omps2 -I$TAMPI_HOME/include

-ltampi test.f90

IFSKernel example

B1. OpenMP tasks + TAMPI

```
#include “TAMPIf.h”
include “mpi.h”
! ...
nreqs=0
tag=1
do proc=1,nprocs
  if (sendoff(proc) > 0) then
    nreqs=nreqs+1
    !$OMP TASK ... DEPEND(IN: senddata(:,proc))
    call TAMPI_Isend(senddata(:,proc), sendoff(proc),
            MPI_REAL8, proc-1, tag, MPI_COMM_WORLD,
            reqs(nreqs), ierr)
    !$OMP END TASK
  endif
  if (recvoff(proc) > 0) then
    nreqs=nreqs+1
    !$OMP TASK ... DEPEND(OUT: recvdata(:,proc), statuses(nreqs))
    call TAMPI_Irecv(recvdata(:,proc), recvoff(proc),
            MPI_REAL8, proc-1, tag, MPI_COMM_WORLD,
            reqs(nreqs), statuses(nreqs), ierr)
    !$OMP END TASK
  endif
enddo
call TAMPI_Waitall(nreqs, reqs(:), statuses(:), ierr)

do proc=1,nprocs
  if (recvoff(proc) > 0) then
    !$OMP TASK ... DEPEND(IN:recvdata(:,proc))
    call process_data(recvdata(:,proc), recvoff(proc))
    !$OMP END TASK
  endif
enddo
! ...
```

mpif90 -fopenmp -I$TAMPI_HOME/include

-ltampi test.f90

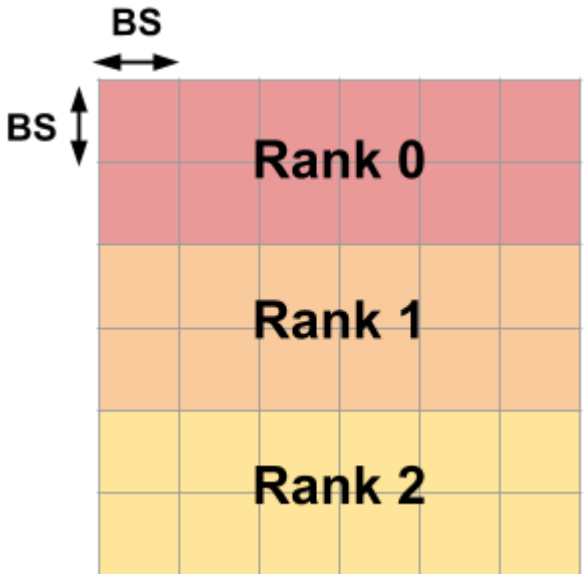
Gauss-Seidel example

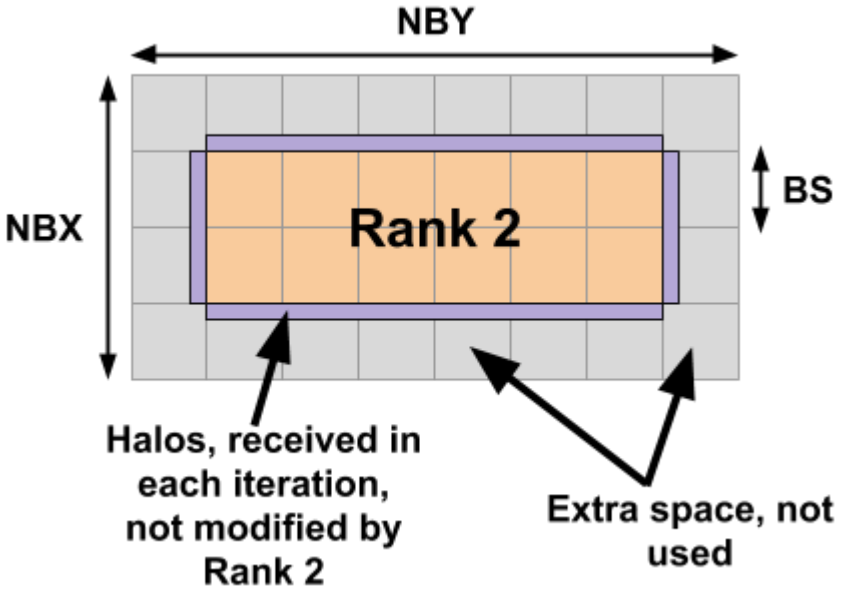
```
for (i=1; i<n; i++)
    for (j=1; j<n; j++) {
        a[i][j] = (a[i-1][j] + a[i][j-1] +
                   a[i][j] + a[i+1][j] + a[i][j+1])/5;
    }
}
```

- **Versions**
  - Pure MPI
  - Hybrid *Fork-join*
  - Hybrid tasks + *sentinel*
  - Hybrid tasks + TAMPI Blocking mode
  - Hybrid tasks + TAMPI Non-blocking mode

Pure MPI (I)

- Ex: 6 x 6 blocks domain, decomposition across 3 MPI ranks





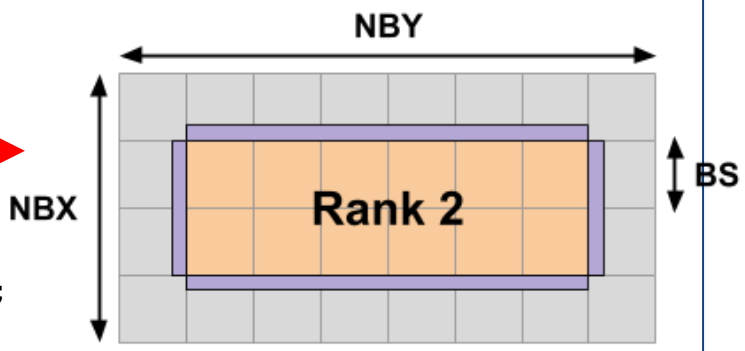
- After each iteration, neighbor MPI ranks have to exchange upper/lower halos

```
typedef double row_t[BS];
typedef row_t block_t[BS]; // blocks are consecutive!
...
block_t matrix[][] = malloc(NBX * NBY * sizeof(block_t));
```

Pure MPI (II)

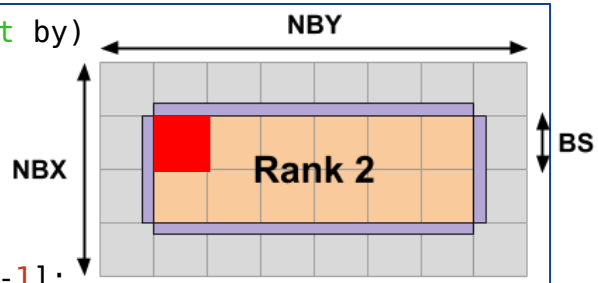
```
void solve(block_t matrix[NBX][NBY], int NBX, int NBY, int timesteps)
{
    int rank, rank_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &rank_size);
    for (int t = 0; t < timesteps; ++t) {
        solveGaussSeidel(matrix, NBX, NBY, rank, rank_size);
    }
    MPI_Barrier(MPI_COMM_WORLD);
}
```

```
void solveGaussSeidel(block_t matrix[NBX][NBY], int NBX, int NBY, int rank, int rank_size)
{
    if (rank != 0) {
        sendFirstComputeRow(matrix, NBX, NBY, rank, rank_size);
        receiveUpperHalo(matrix, NBX, NBY, rank, rank_size);
    }
    if (rank != rank_size - 1) {
        receiveLowerHalo(matrix, NBX, NBY, rank, rank_size);
    }
    for (int bx = 1; bx < NBX-1; ++bx) {
        for (int by = 1; by < NBY-1; ++by) {
            solveBlock(matrix, NBX, NBY, bx, by);
        }
    }
    if (rank != rank_size - 1) {
        sendLastComputeRow(matrix, NBX, NBY, rank, rank_size);
    }
}
```

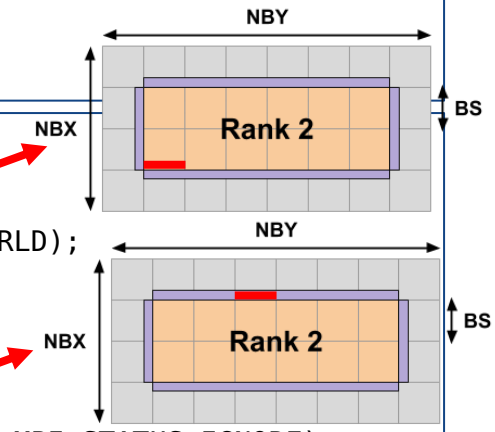


Pure MPI (III)

```
void solveBlock(block_t matrix[NBX][NBY], int NBX, int NBY, int bx, int by)
{
    block_t &centerBlock = matrix[bx][by];
    const block_t &topBlock = matrix[bx-1][by];
    const block_t &bottomBlock = matrix[bx+1][by];
    ...
    for (int x = 0; x < BS; ++x) {
        const row_t &topRow = (x > 0) ? centerBlock[x-1] : topBlock[BS-1];
        const row_t &bottomRow = (x < BS-1) ? centerBlock[x+1] : bottomBlock[0];

        for (int y = 0; y < BS; ++y) {
            double left = (y > 0) ? centerBlock[x][y-1] : leftBlock[x][BS-1];
            double right = (y < BS-1) ? centerBlock[x][y+1] : rightBlock[x][0];
            centerBlock[x][y] = 0.25 * (topRow[y] + bottomRow[y] + left + right);
        }
    }
}
```

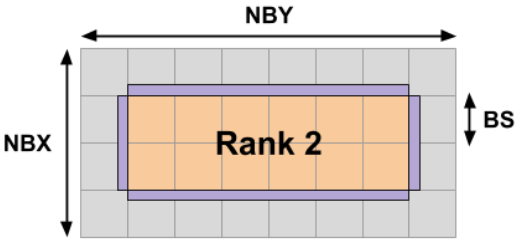



```
void sendLastComputeRow(block_t matrix[NBX][NBY], int NBX, int NBY, int dst)
{
    for (int by = 1; by < NBY-1; ++by) {
        MPI_Send(&matrix[NBX-2][by][BS-1], BS, MPI_DOUBLE, dst, by, MPI_COMM_WORLD);
    }
}
void receiveUpperHalo(block_t matrix[NBX][NBY], int NBX, int NBY, int src)
{
    for (int by = 1; by < NBY-1; ++by) {
        MPI_Recv(&matrix[0][by][BS-1], BS, MPI_DOUBLE, src, by, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

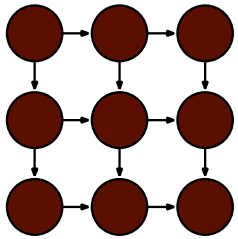


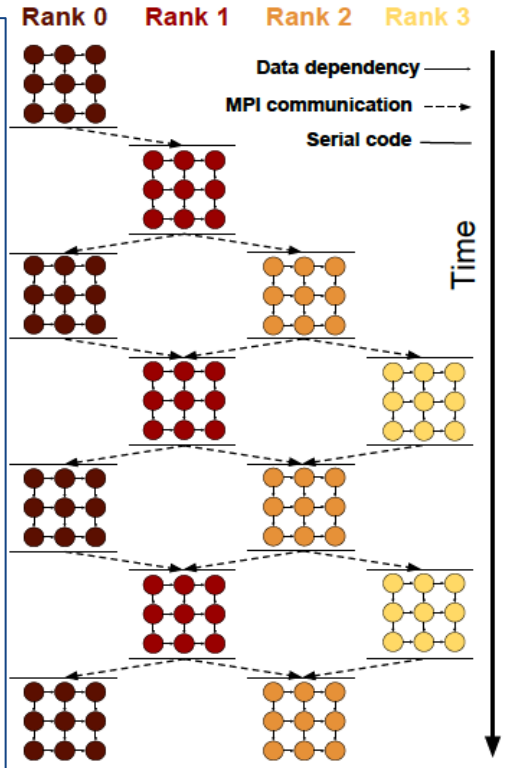
Pure MPI (IV)

- Fully sequential execution
- **No overlapping** of communication and computation phases
- **One rank per core**




```
void solveGaussSeidel(block_t matrix[NBX][NBY], int NBX, int NBY, int rank, int rank_size)
{
    if (rank != 0) {
        sendFirstComputeRow(matrix, NBX, NBY, rank-1);
        receiveUpperHalo(matrix, NBX, NBY, rank-1);
    }
    if (rank != rank_size - 1) {
        receiveLowerHalo(matrix, NBX, NBY, rank+1);
    }
    for (int bx = 1; bx < NBX-1; ++bx) {
        for (int by = 1; by < NBY-1; ++by) {
            solveBlock(matrix, NBX, NBY, bx, by);
        }
    }

    if (rank != rank_size - 1) {
        sendLastComputeRow(matrix, NBX, NBY, rank+1);
    }
}
```

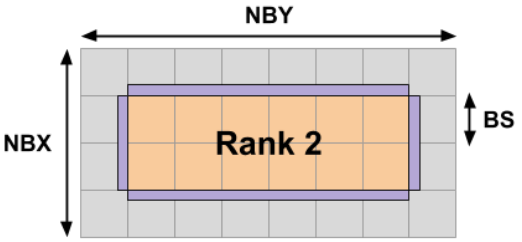




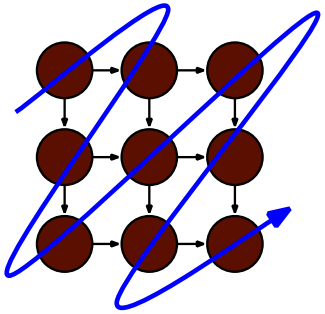
1 Rank x Core!!!

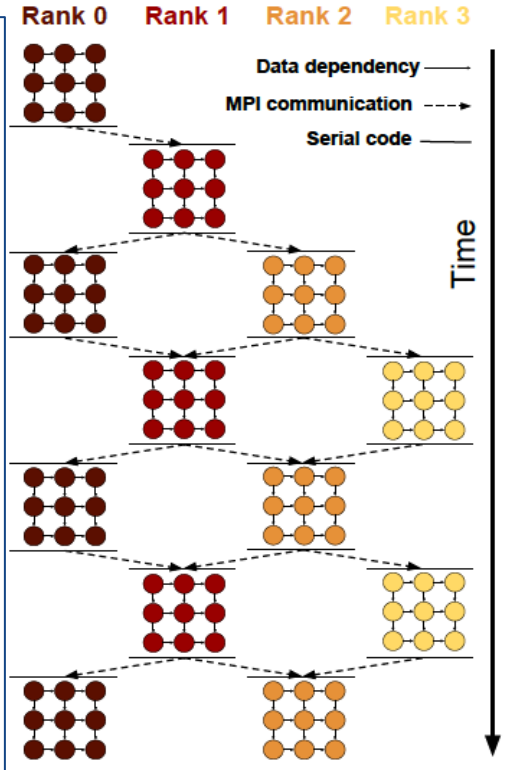
Fork-Join

- OmpSs-2 used to execute in **parallel** the **computation** phases
- **MPI** used only on **sequential** phases for communications
- **No overlapping** of communication and computation phases




```
void solveGaussSeidel(block_t matrix[NBX][NBY], int NBX, int NBY, int rank, int rank_size)
{
    if (rank != 0) {
        sendFirstComputeRow(matrix, NBX, NBY, rank-1);
        receiveUpperHalo(matrix, NBX, NBY, rank-1);
    }
    if (rank != rank_size - 1) {
        receiveLowerHalo(matrix, NBX, NBY, rank+1);
    }
    for (int bx = 1; bx < NBX-1; ++bx) {
        for (int by = 1; by < NBY-1; ++by) {
                #pragma oss task                \
                  in(matrix[bx-1][by]) \
                  in(matrix[bx][by-1]) \
                  in(matrix[bx][by+1]) \
                  in(matrix[bx+1][by]) \
                  inout(matrix[bx][by])
            solveBlock(matrix, NBX, NBY, bx, by);
        }
    }
    #pragma oss taskwait
    if (rank != rank_size - 1) {
        sendLastComputeRow(matrix, NBX, NBY, rank+1);
    }
}
```

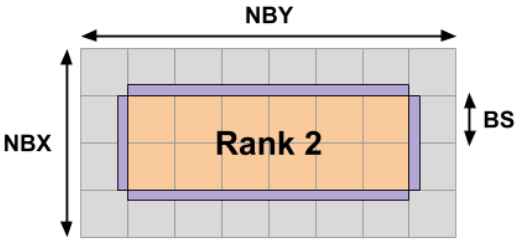




1 Rank x Node!!!

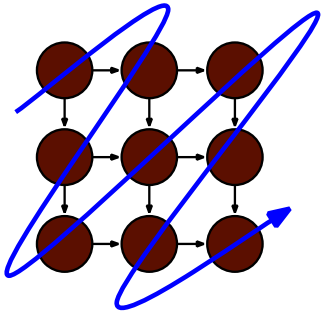
Tasks + sentinel (I)

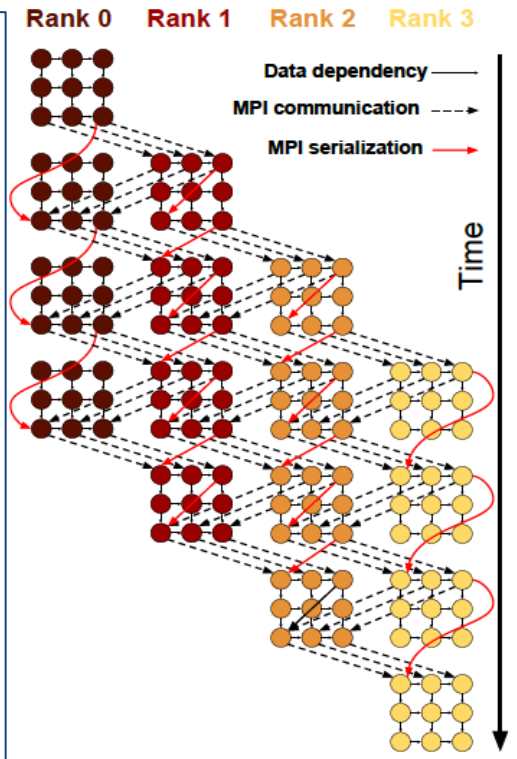
- Tasks used for both computations and communications
- **Tags** used to match send and receive operations but ...
- Communication tasks have to be **serialized** to avoid **deadlocks**
- **Partial overlapping** of communication and computation phases




```
void solveGaussSeidel(block_t matrix[NBX][NBY], int NBX, int NBY, int rank, int rank_size)
{
    if (rank != 0) {
        sendFirstComputeRow(matrix, NBX, NBY, rank-1);
        receiveUpperHalo(matrix, NBX, NBY, rank-1);
    }
    if (rank != rank_size - 1) {
        receiveLowerHalo(matrix, NBX, NBY, rank+1);
    }
    for (int bx = 1; bx < NBX-1; ++bx) {
        for (int by = 1; by < NBY-1; ++by) {
                #pragma oss task                \
                  in(matrix[bx-1][by]) \
                  in(matrix[bx][by-1]) \
                  in(matrix[bx][by+1]) \
                  in(matrix[bx+1][by]) \
                  inout(matrix[bx][by])
            solveBlock(matrix, NBX, NBY, bx, by);
        }
    }

    if (rank != rank_size - 1) {
        sendLastComputeRow(matrix, NBX, NBY, rank+1);
    }
}
```

No taskwait!

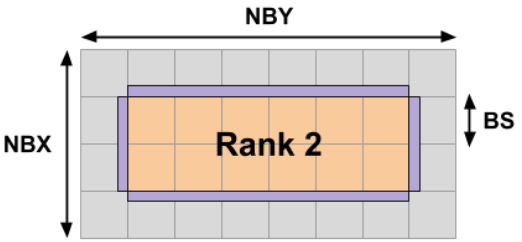




1 Rank x Node!!!

Tasks + sentinel (II)

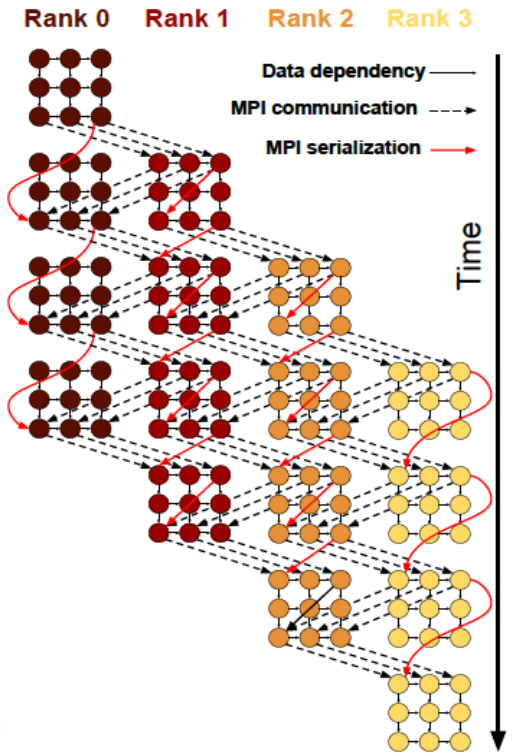
- Tasks used for both computations and communications
- The **same tag (1)** is used for all communications so ...
- Communication tasks have to be **serialized** to avoid **deadlocks**
- **Partial overlapping** of communication and computation phases




```
void sendLastComputeRow(block_t matrix[NBX][NBY], int NBX, int NBY, int dst)
{
    for (int by = 1; by < NBY-1; ++by) {
        #pragma oss task in(matrix[NBX-2][by]) inout(serial)
        MPI_Send(&matrix[NBX-2][by][BS-1], BS, MPI_DOUBLE, dst, 1,
                                    MPI_COMM_WORLD);
    }
}

void receiveUpperHalo(block_t matrix[NBX][NBY], int NBX, int NBY, int src)
{
    for (int by = 1; by < NBY-1; ++by) {
        #pragma oss task out(matrix[0][by]) inout(serial)
        MPI_Recv(&matrix[0][by][BS-1], BS, MPI_DOUBLE, src, 1,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

Sentinel to serialize communication tasks

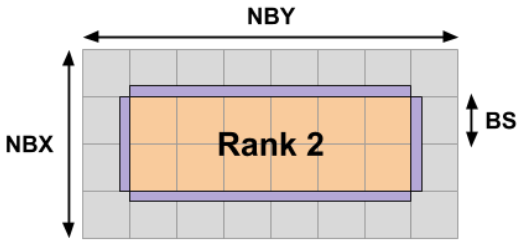
MPI_THREAD_MULTIPLE



1 Rank x Node!!!

Tasks + sentinel (III)

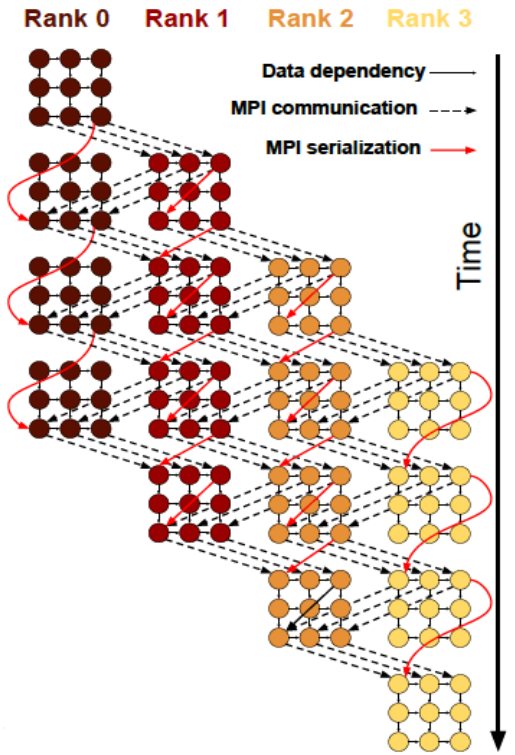
- Tasks used for both computations and communications
- **Tags (block id)** used to match communications but still ...
- Communication tasks have to be **serialized** to avoid **deadlocks**
- **Partial overlapping** of communication and computation phases




```
void sendLastComputeRow(block_t matrix[NBX][NBY], int NBX, int NBY, int dst)
{
    for (int by = 1; by < NBY-1; ++by) {
        #pragma oss task in(matrix[NBX-2][by]) inout(serial)
        MPI_Send(&matrix[NBX-2][by][BS-1], BS, MPI_DOUBLE, dst, by,
                                    MPI_COMM_WORLD);
    }
}

void receiveUpperHalo(block_t matrix[NBX][NBY], int NBX, int NBY, int src)
{
    for (int by = 1; by < NBY-1; ++by) {
        #pragma oss task out(matrix[0][by]) inout(serial)
        MPI_Recv(&matrix[0][by][BS-1], BS, MPI_DOUBLE, src, by,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

Sentinel to serialize communication tasks

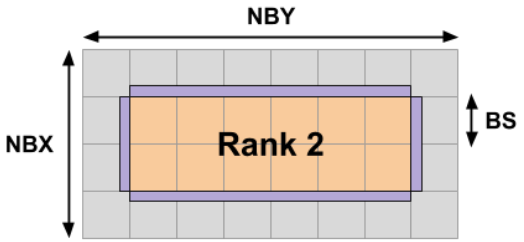
MPI_THREAD_MULTIPLE



1 Rank x Node!!!

Tasks + TAMPI: Blocking Mode

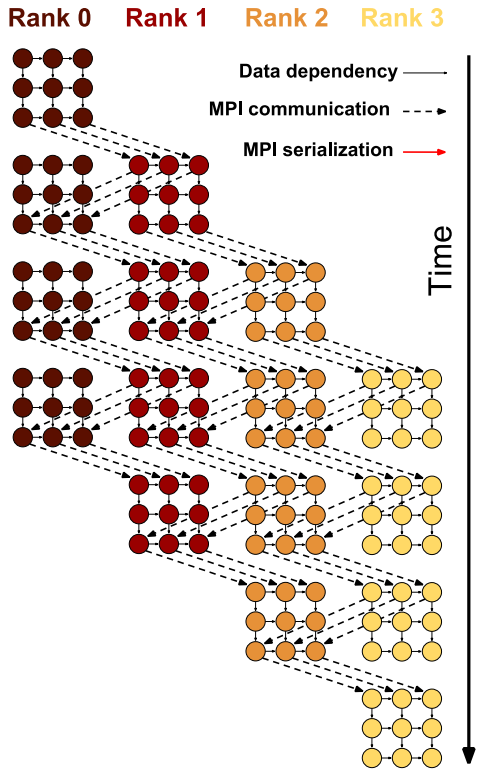
- Tasks used for both **computations** and **communications**
- **Tags** used to match send and receive operations
- **Full overlapping** of communication and computation phases




```
void sendLastComputeRow(block_t matrix[NBX][NBY], int NBX, int NBY, int dst)
{
    for (int by = 1; by < NBY-1; ++by) {
        #pragma oss task in(matrix[NBX-2][by])
        MPI_Send(&matrix[NBX-2][by][BS-1], BS, MPI_DOUBLE, dst, by,
                                    MPI_COMM_WORLD);
    }
}


void receiveUpperHalo(block_t matrix[NBX][NBY], int NBX, int NBY, int src)
{
    for (int by = 1; by < NBY-1; ++by) {
        #pragma oss task out(matrix[0][by])
        MPI_Recv(&matrix[0][by][BS-1], BS, MPI_DOUBLE, src, by,
                                    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }
}
```

No sentinel needed!

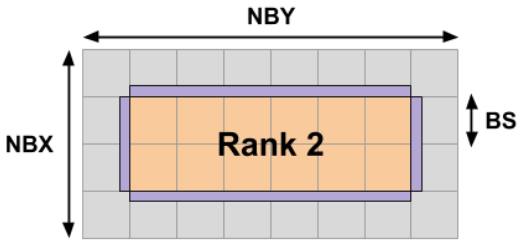
MPI_TASK_MULTIPLE!



1 Rank x Node!!!

Tasks + TAMPI: Non-Blocking Mode

- Tasks used for both **computations** and **communications**
- **Tags** used to match send and receive operations
- **Full overlapping** of communication and computation phases

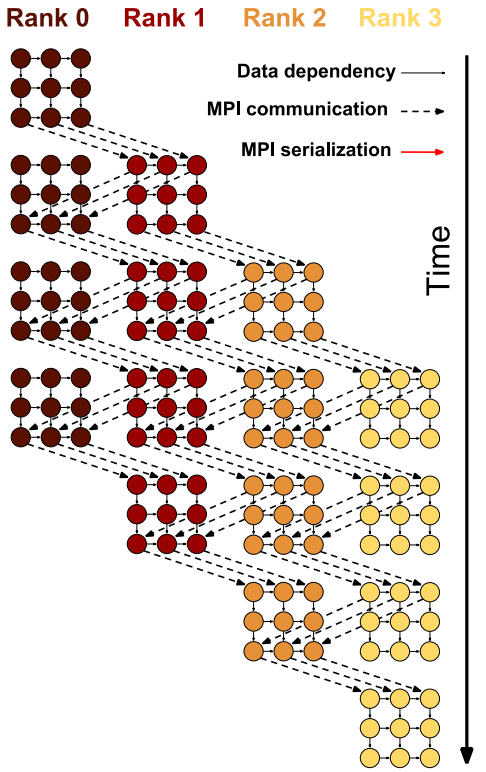



```
void sendLastComputeRow(block_t matrix[NBX][NBY], int NBX, int NBY, int dst)
{
    for (int by = 1; by < NBY-1; ++by) {
        #pragma oss task in(matrix[NBX-2][by])
        {
          MPI_Request request;
          MPI_Isend(&matrix[NBX-2][by][BS-1], BS, MPI_DOUBLE, dst, by,
                                  MPI_COMM_WORLD, &request);
          TAMPI_Iwait(&request, MPI_STATUS_IGNORE);
        }
    }
}
```

OR

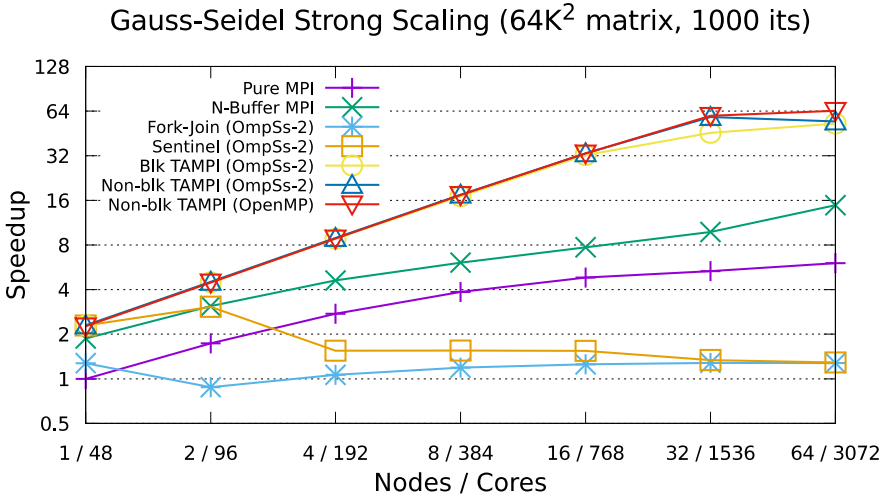
```
void sendLastComputeRow(block_t matrix[NBX][NBY], int NBX, int src)
{
    for (int by = 1; by < NBY-1; ++by) {
        #pragma oss task in(matrix[NBX-2][by])
        {
          MPI_Request request;
          TAMPI_Isend(&matrix[NBX-2][by][BS-1], BS, MPI_DOUBLE, src, by,
                                  MPI_COMM_WORLD, &request);
        }
    }
}
```

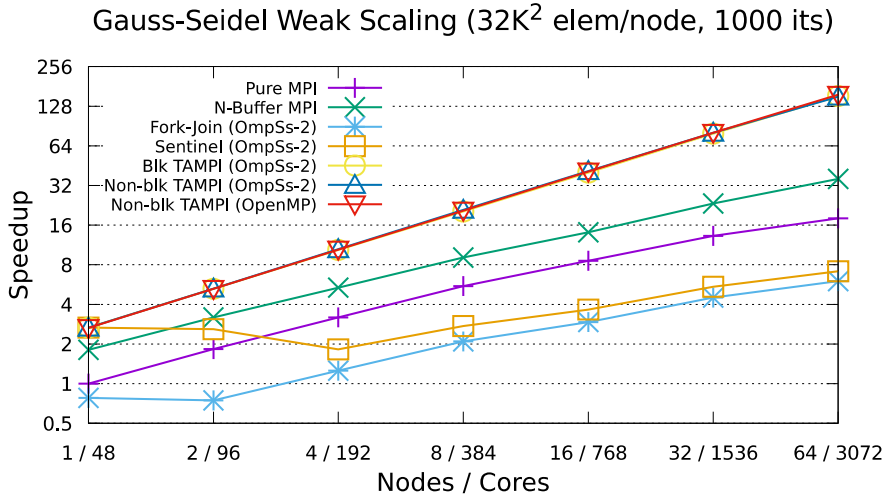
MPI_THREAD_MULTIPLE!

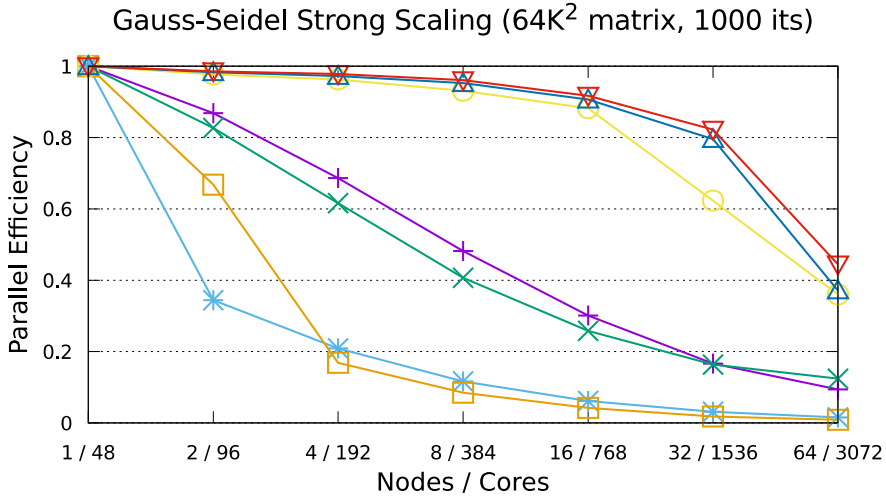


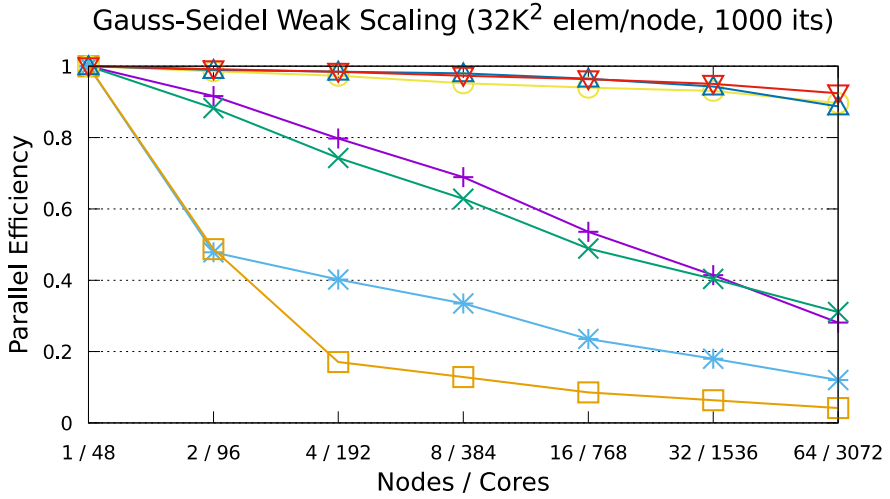
1 Rank x Node!!!

Tiled Gauss-Seidel (MN4, 48 cores x Node)





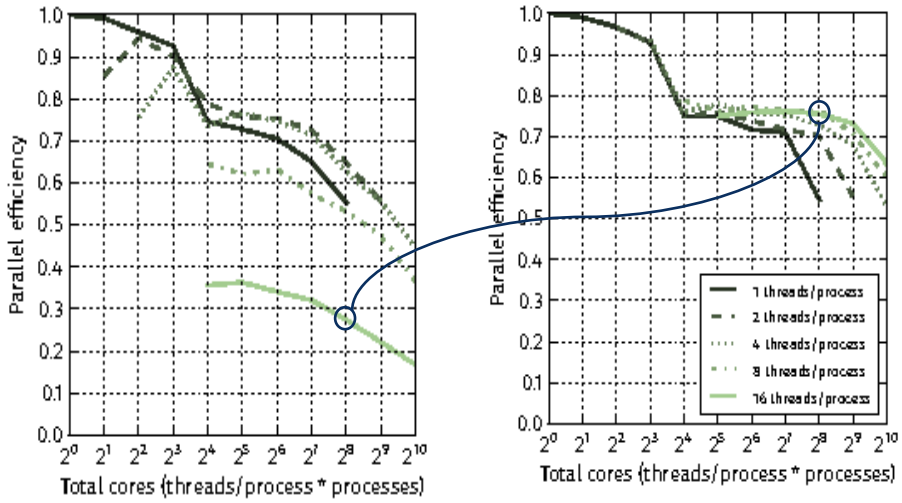




Taskifying communications

```
1:    RIMP2_RMP2Energy_InCore_V_MPIOMP ()
          …
405:      DO LNumber_Base
498:          DGEMM
              …
518:          if (something)
                  {  wait ;          // for current iter.
                     Isend, Irecv;   // for next iter.
                  }
              allreduce
588:          Do loops
                      Evaluating MP2 correlation
636:          END DO
          ENDO
```

NTCHEM



Don’t mask your symptoms

Top down

Leave order (overlap) to OpenMP

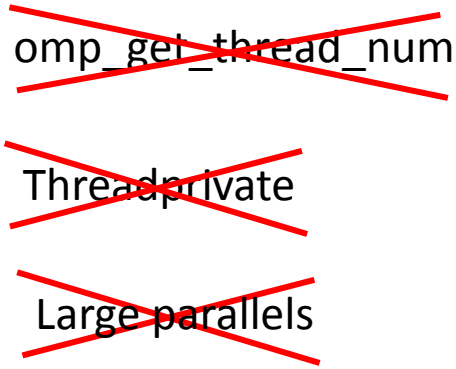
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Malleability

Malleability

- Application structure not tied to initially allocated resources

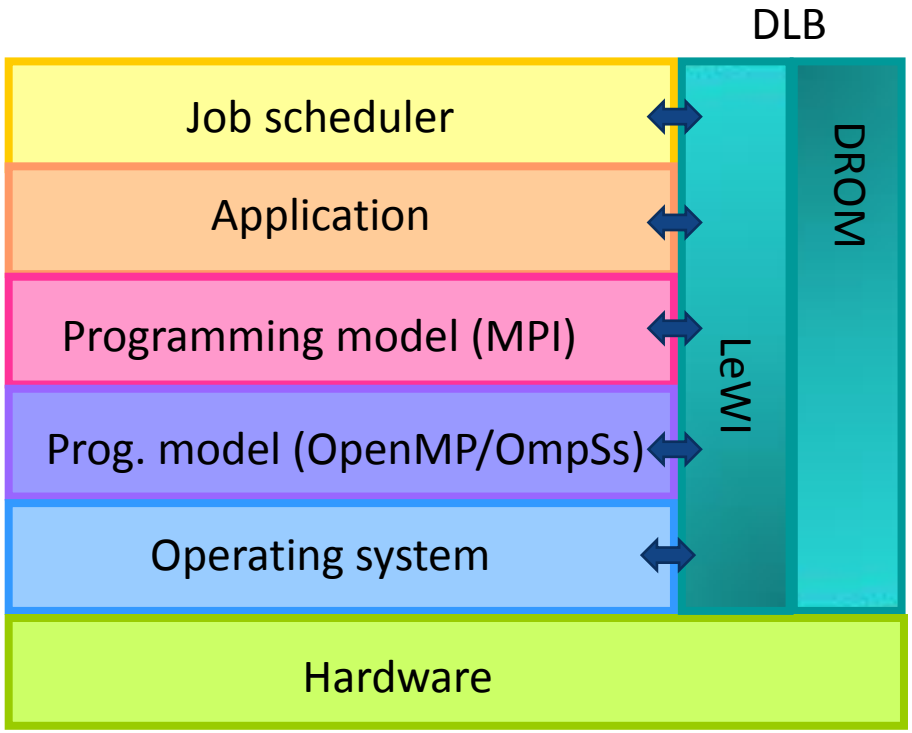


- Can adapt dynamically to increase/reduction of resources

- Impacts programming model and practices !!!!

Dynamic resource allocation

- Coordination of resource usage across the different levels
  - DROM (Dynamic Resource Ownership Manager) at medium/coarse granularity (>10s)
  - DLB (Dynamic Load balancing Library) at fine granularity (>10s of microseconds)



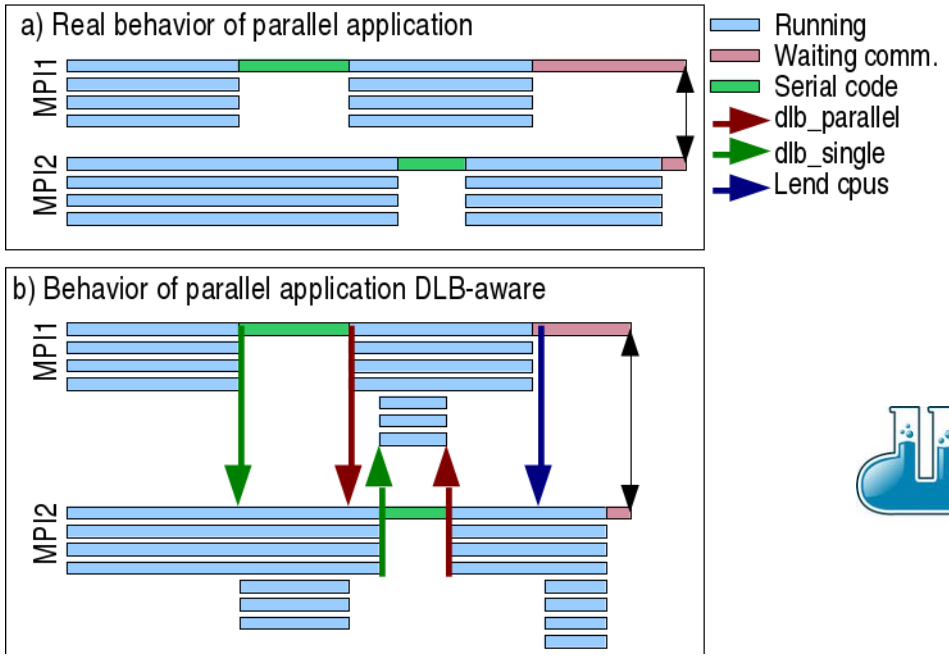
Terminology

- CPU: execution resource
  - CPUid [0..N]: Identifier of a CPU within a node
- Thread:
  - Execution context for a sequential control flow
  - Frequently mapped 1:1 to a CPU
- Cpuset:
  - Set of cores where a Linux thread may run
  - Mask: data type listing a set of CPUids
  - Typically:
    - CPU set for all the threads in each process assigned at launch time
    - Homogeneous in size across all processes in an MPI job
      - OMP_NUM_THREADS
    - Never changes

DLB: Dynamic Load balancing concepts

- Two allocation concepts
  - Ownership / owned set
    - A process owns a CPU and has priority to use it
    - A CPU can only be owned by one process at most
    - A given owned set is assigned to a process at launch time
    - Mechanism to managed/changed by DROM.
      - Target Dynamicity granularities >~10ms
    - Actual policy implemented in externa resource manager
      - Integrated in SLURM, ….
  - Active cpuset
    - Set of CPUS a process can use at given point in time
    - Managed/changed by LeWI
      - Implements policies
      - Target Dynamicity granularities >~1ms

DLB: Lend when Idle (LeWI)



dynamic load balancing

- Opportunity to fight Amdalh's law
  - Productive / Easy !!!
    - Nx1
    - Hybridize imbalanced regions

"LeWI: A Runtime Balancing Algorithm for Nested Parallelism". M.Garcia et al. ICPP09
"Hints to improve automatic load balancing with LeWI for hybrid applications" JPDC2014

DLB: Lend when Idle (LeWI)

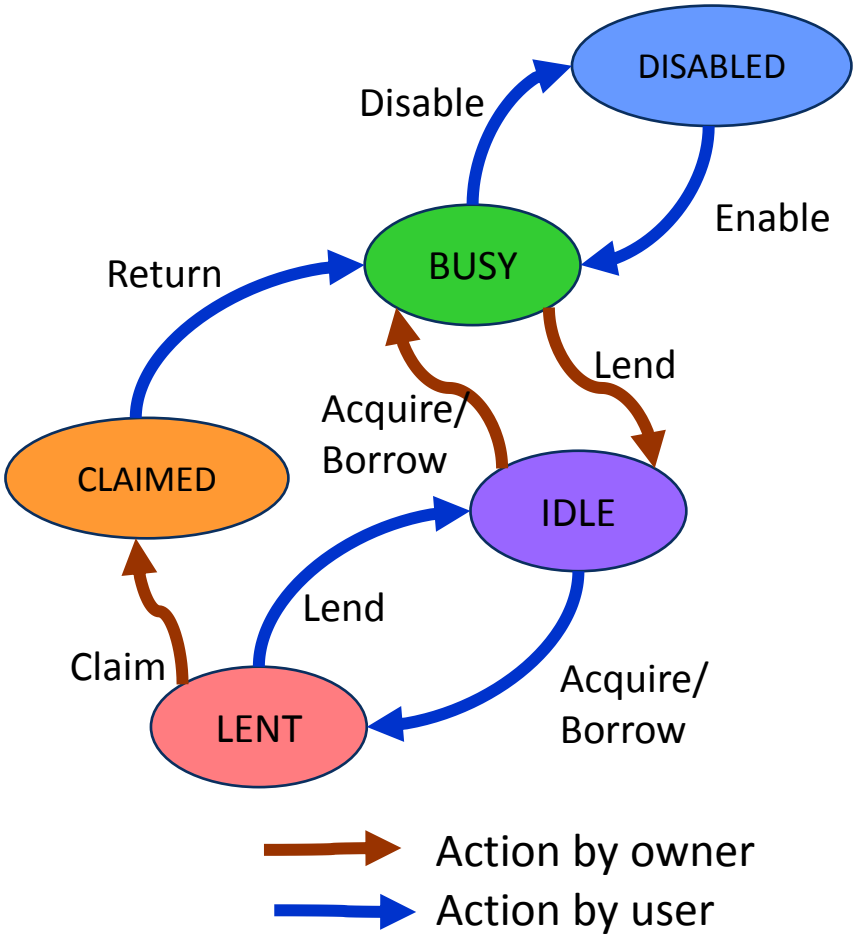
- Library linked to each process in the node.
- DLB can change the active set, implementing policies coordinating the different processes in a node
- A process may **enable** DLB, committing to follow the coordination practices below
- A process may **disable** DLB → Its active set goes back to its owned set.
- An enabled DLB process may request its active set to change at fine granularity
  - **Lend**ing CPUs if it "knows/expects" it will not need them for some time.
  - **Borrow**ing CPUs beyond its default allocation if it considers will be able to efficiently use more cores if allocated.
  - **Claim**ing back cores it owns and had previously lent
  - DLB may fully or partially honor those requests based on its knowledge of the overall request of DLB activated processes within the node and the DLB policies.
  - To serve a process request, DLB may have to interact with other processes claiming back some CPUs form their active set
  - DLB will notifies its decision to the process
- A process has preference to use its owned CPUs.
- A process has to "frequently" monitor claim back requests (**Return_cpus**) and honor by releasing the core (immediately / as soon as possible).
- Transitions releasing a core require the process it to suspend the thread that was running on it (done by DLB invoking the Programming model runtime)
- Transitions acquiring cores require it to resume the thread (done by DLB invoking the Programming model runtime)

DLB: Lend when Idle (LeWI)

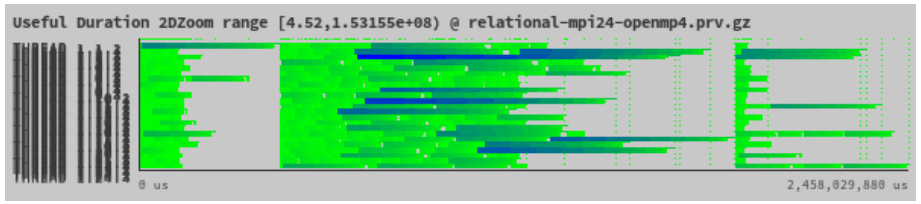
- MPI interface
  - @ MPI blocking calls
    - Release all cores
    - Release all but the OpenMP master thread
- OmpSs: within the Nanos runtime
  - Ready queue gets too empty …
  - … or to full
- OpenMP OMPT
  - borrow at the beginning of parallels, return to default at the end.
- DLB API: explicit programmer hints

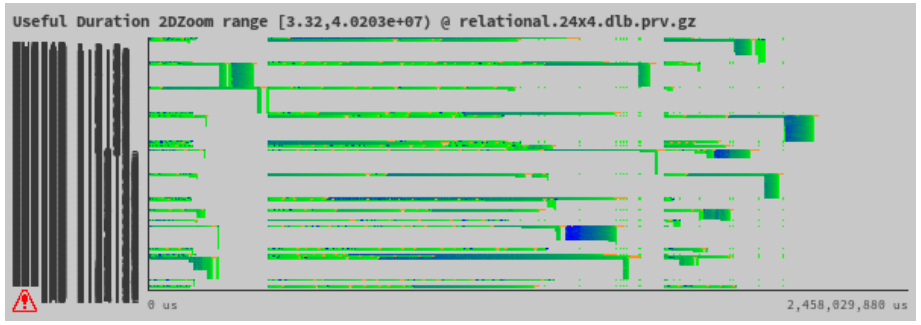
https://pm.bsc.es/ftp/dlb/doc/user-guide/api.html

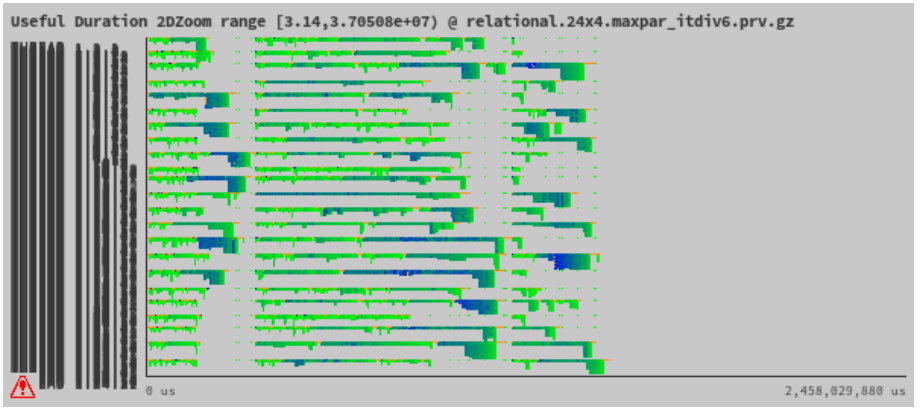
CPU state diagram



Example: Relational Discovery



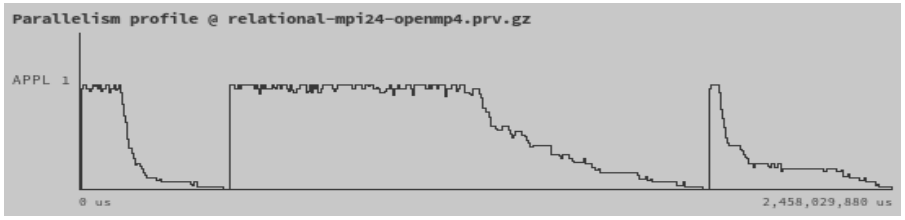


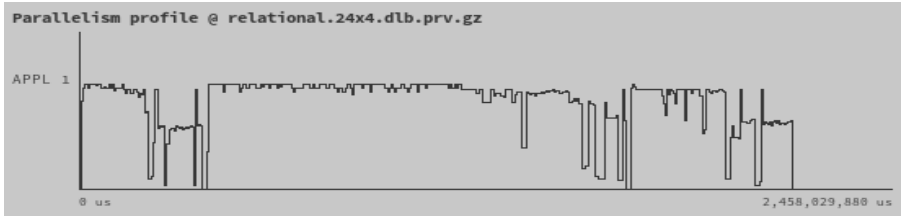


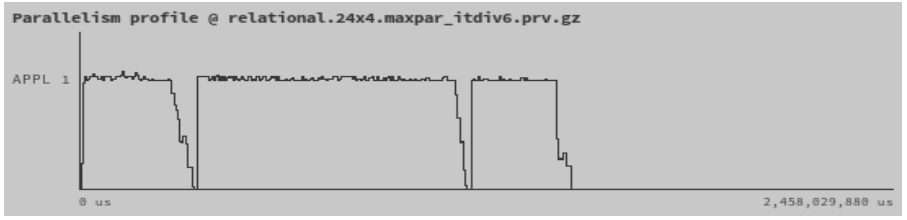
dynamic load balancing

Relational Discovery

Example: Relational Discovery







dynamic load balancing

Relational Discovery

"LeWI: A Runtime Balancing Algorithm for Nested Parallelism". M.Garcia et al. ICPP09
"Hints to improve automatic load balancing with LeWI for hybrid applications" JPDC2014

Examples

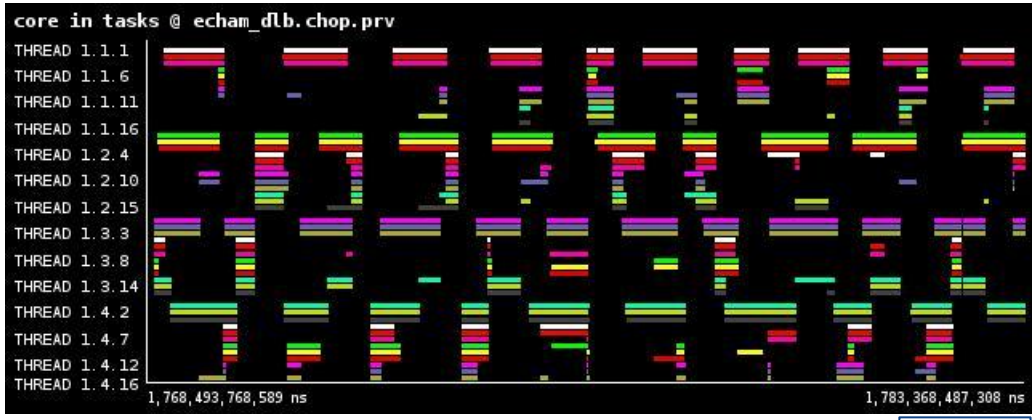

Parallel functions_2DZoom_range[13.00,34.00] @ bt-mz.D.32x32.
Parallel functions_2DZoo
BT-MZ


Task @ trace-GROMACS.16x4.SMPSS-DYN-MPI.LeWI.BLOCK.chop2.shifted.prv
GROMACS

Fighting the OS

- Coordinating processes within node
  - Through a shared memory region
  - User mode implementation → non atomicity
  - Explicit pinning of threads and handoff scheduling (Fighting the Linux kernel)

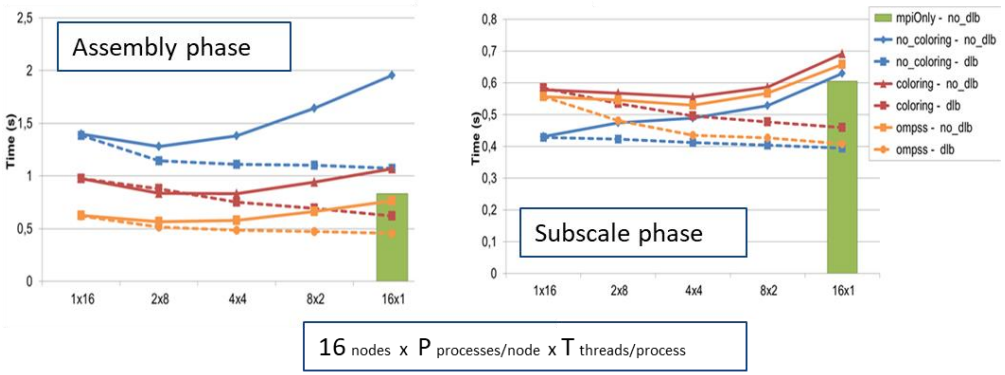
- Desirable Kernel support for atomic handoff

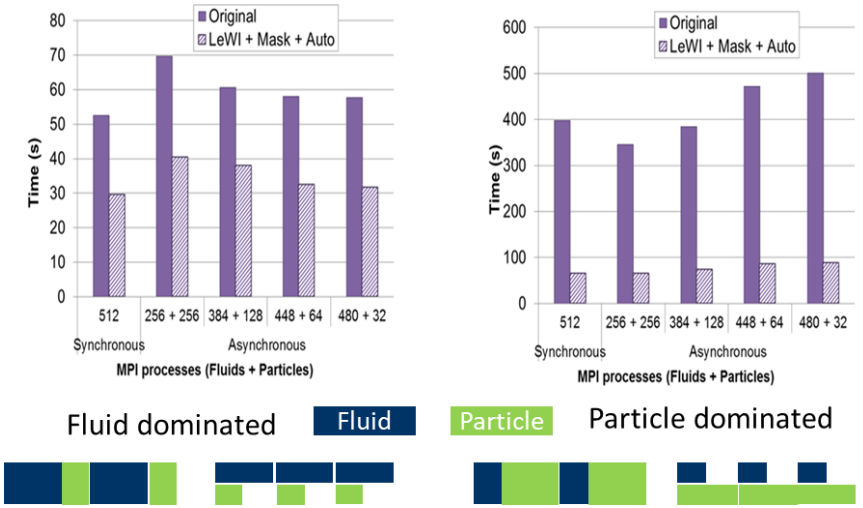


ECAM

Towards the throughput age

- Dynamic resource sharing/management
- Configuration independence
  - Amount of resources is what really matters
- Side effects
  - Nx1 can be better than pure MPI !!!
  - hope for lazy programmers



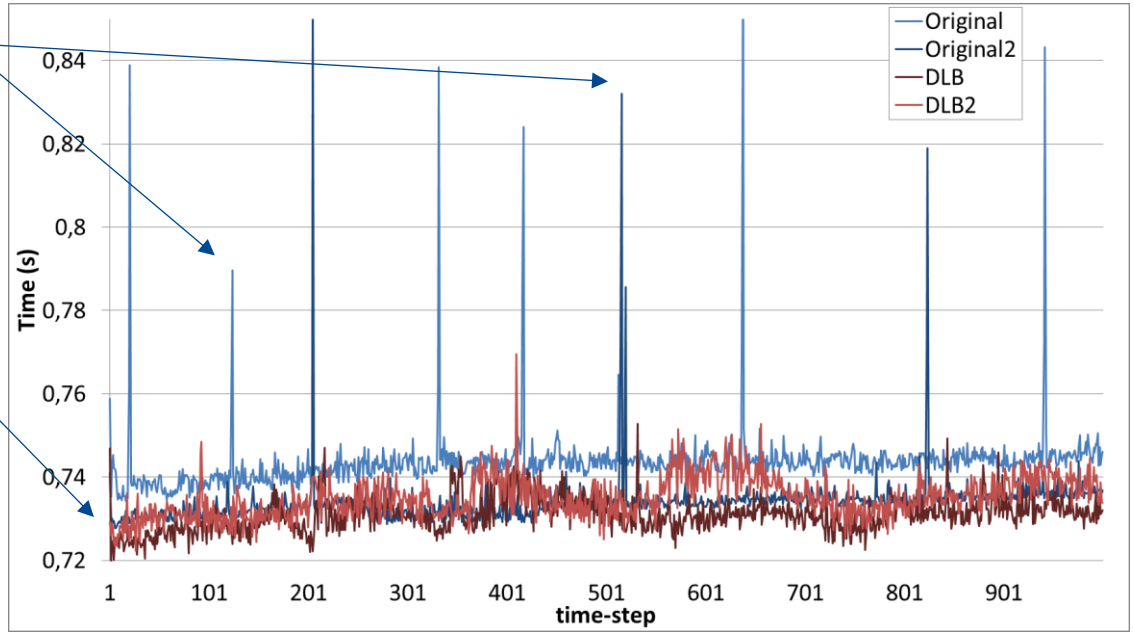


Reacting to noise

- Even if application is balanced, OS noise may introduce variability …
- …that can be absorbed by DLB

Blue lines original application (2 different runs)
Red lines same run with DLB (2 different runs)

Clearly visible spikes without DLB are absorbed by DLB



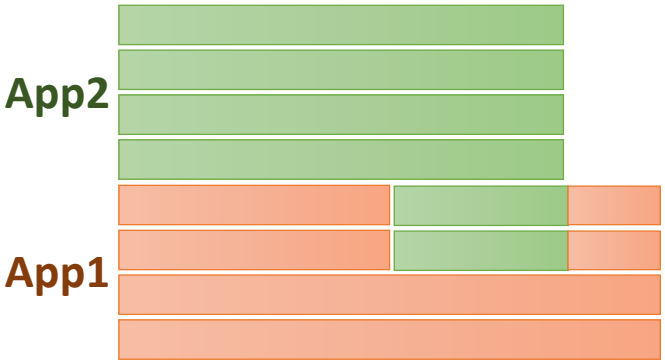
DROM

- Library to offer mechanism to an external resource manager to dynamically change the ownership of cores between processes.

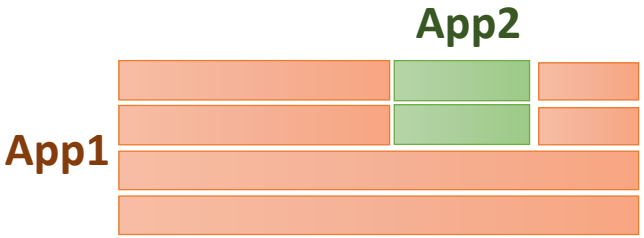
- DROM can change the default allocation of connected processes
  - coordinating them to avoid oversubscription
  - Applying policies based on the observed behavior of the application

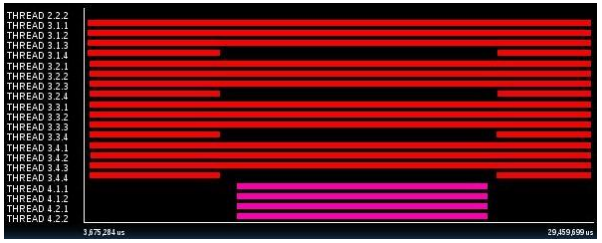
DROM: Use cases

- A) User:
  Increase priority to App2

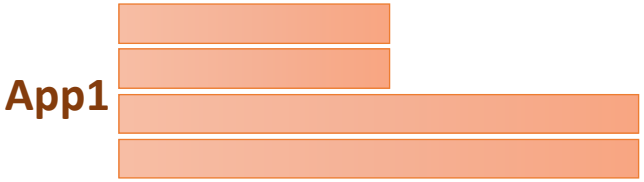


- B) Job Scheduler:
  Run High priority App2 in resources assigned to App1





- C) App1: Release 2 CPUs because not using efficiently



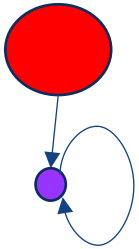
Barcelona Supercomputing Center
Centro Nacional de Supercomputación

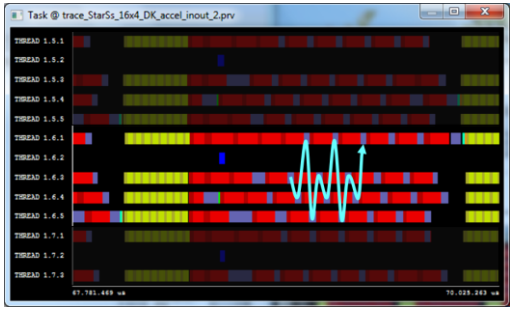
EXCELENCIA SEVERO OCHOA

Beware of reductions on large arrays

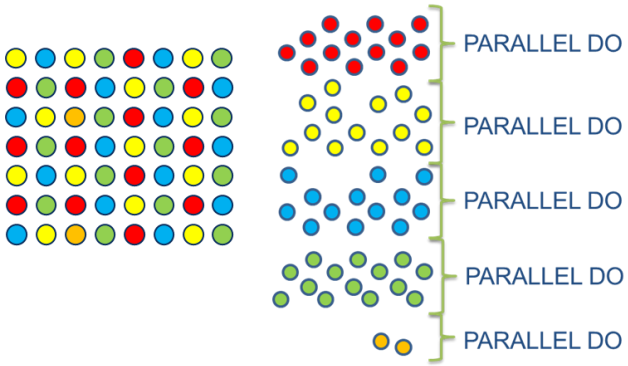
Beware of reductions on large arrays

- Reductions with indirection on large arrays with indirections
  - Atomic
    - Contention & overhead
  - Array privatization + final reduction
    - High memory consumption and additional operations count
  - Serialize
    - Commutative clause





  - Coloring
    - Locality, IPC issues



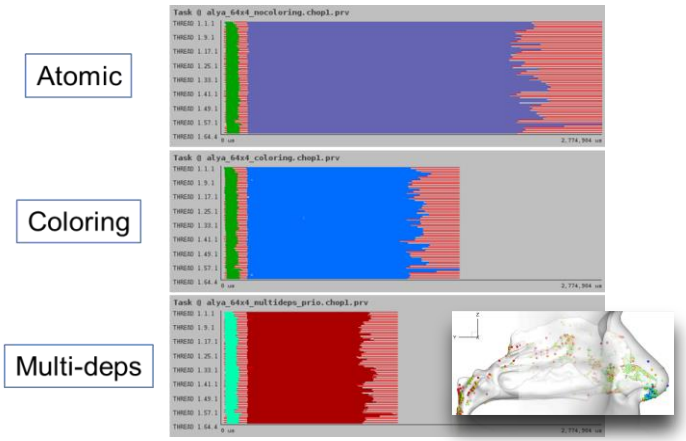
Multidependences

- 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
- Multi-dependences
  - Syntax to specify a dynamic number of directionality clauses

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

- Supported in OpenMP 5.0 by iterators mechanism (more generic)

```
for (int i = 0; i < N; ++i) {
   int nb = neigh[i].size();
#pragma omp task depend(out:X[i]) \
depend(iterator(j=0,nb), in:X[neigh[i][j]])
   RebuildGridMT(X[i]);
}
```



Commutative

```
#pragma omp task commutative (lvalue_expr)
```

- Relaxed inout directionality clause
  - Enables the scheduler to change the order (but **not execute concurrently**) the tasks within the inout chains built by the concurrent clause.
    - Dependences resulting form the regular in, out and inout clauses towards and from the concurrent chain are honoured for all tasks in the concurrent chain.

→ OpenMP 5.0

```
#pragma omp task depend (mutexinoutset: var)
```

Commutative

```
#pragma omp task commutative (lvalue_expr)
```

- Relaxed inout directionality clause
  - Enables the scheduler to change the order (but **not execute concurrently**) the tasks within the inout chains built by the concurrent clause.
    - Dependences resulting form the regular in, out and inout clauses towards and from the concurrent chain are honoured for all tasks in the concurrent chain.

→ OpenMP 5.0

```
#pragma omp task depend (mutexinoutset: var)
```

Concurrent

```
#pragma omp task concurrent (lvalue_expr)
```

- Relaxed inout directionality clause
  - Enables the scheduler to change the order (or even execute concurrently) the tasks within the inout chains built by the concurrent clause.
    - Dependences resulting form the regular in, out and inout clauses towards and from the concurrent chain are honoured for **all** tasks in the concurrent chain.
  - The programmer assumes responsibility to handle potential races within the concurrent chain using if needed OpenMP pragmas

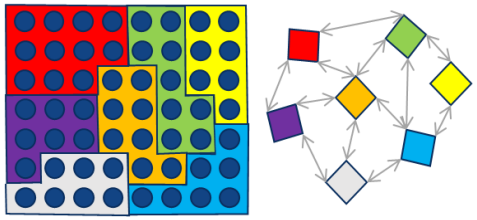
```
#pragma omp atomic
#pragma omp critical
```

→ OpenMP 5.1

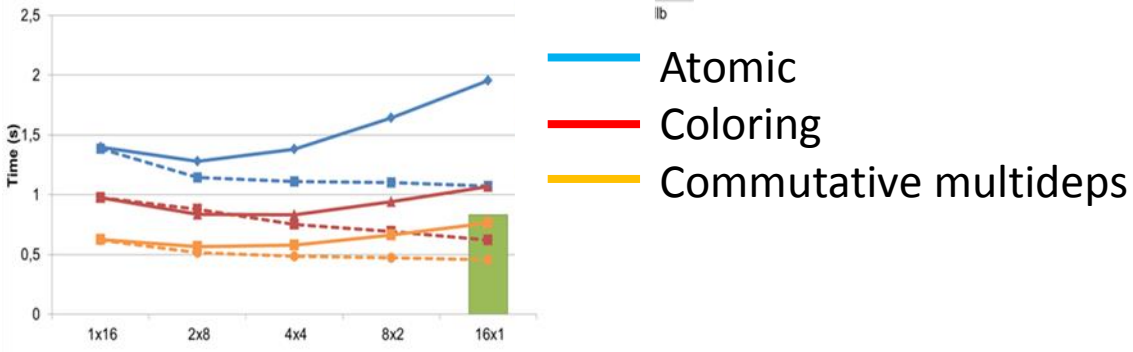
```
#pragma omp task depend (inoutset: var)
```

Finite Element codes (Alya)

- Reductions with indirection on large arrays with indirections
  - Specify incompatibilities !!!
    - Commutative + multidependences



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



Alya

16 nodes x P processes/node x T threads/process

DMRG structure

- Density Matrix Renormalization Group app in condensed matter physics (ORNL)

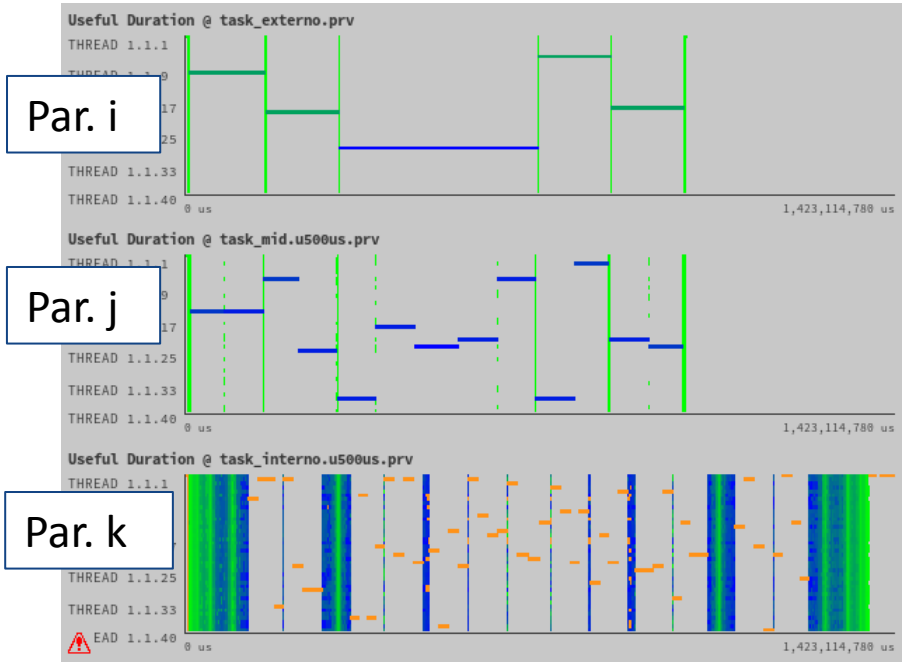
- Skeleton
  - 3 nested loop
  - Reduction on large array
  - Huge variability of op cost

- Real miniapp
  - Different sizes of Y entries

```
T Y[N];

for (i)
  for (j)
    for (k)
      Y[i] += M[k] op X[j]
```

OpenMP parallelizations

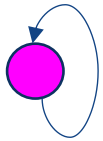
- OpenMP parallelizations
  - Reduction
    - Based on full array privatization
    - Using reduction clauses
  - Nested parallels
    - Worksharings / Tasks
    - Synchronization at end of parallels exposes cost of load imbalances at all levels
    - Overheads at fine levels
    - Issues activation of multiple levels
      - Core partition
      - Levels of Privatization



Taskification

- Serialize reductions
  - Multiple dependence chains

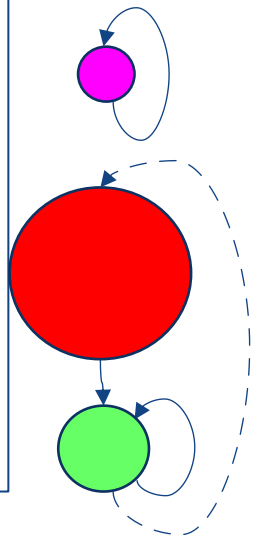
```
T Y[N];


for (i)
  for (j)
    for (k)

            Y[i] +=M[k] op X[j]
```



Taskification

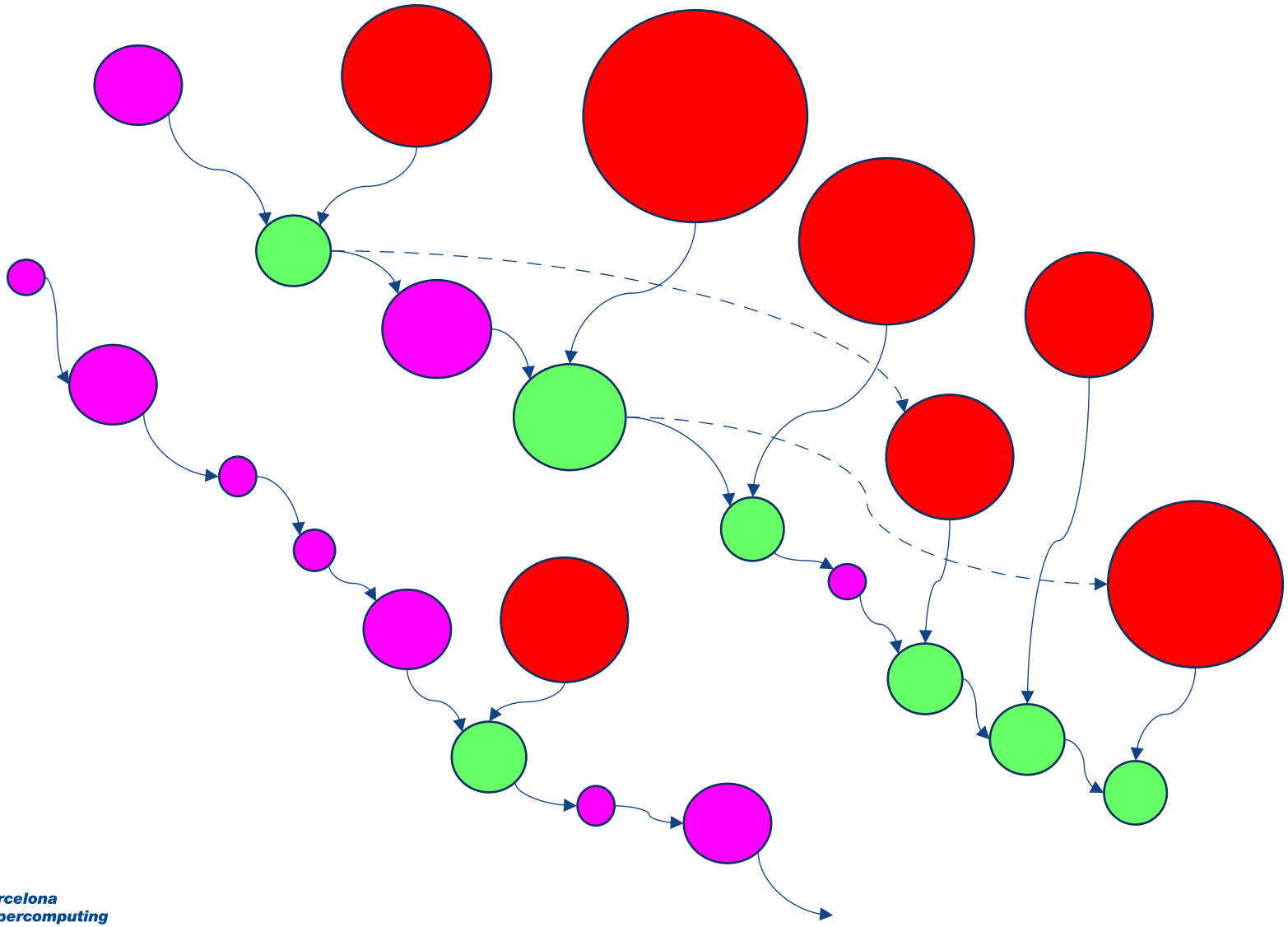
- Serialize reductions
  - Multiple dependence chains
- Split operations
  - Compute & reduce
  - Persist intermediate result
    - Global array of tmps
    - Used in a circular way
    - Enforce antidependence

```
T Y[N];
T tmp[Npriv];

for (i)
  for (j)
    for (k)
      if (small)
            Y[i] +=M[k] op X[j]

       else

            tmp[next]=M[k] op X[j]


            Y[i] += tmp[next];
```



- Reduce overhead → do not split small operation
  - Compute directly on target operand
  - Avoid task instantiation and dependence overhead
  - Avoid memory allocation, initialization & reduction to target operand

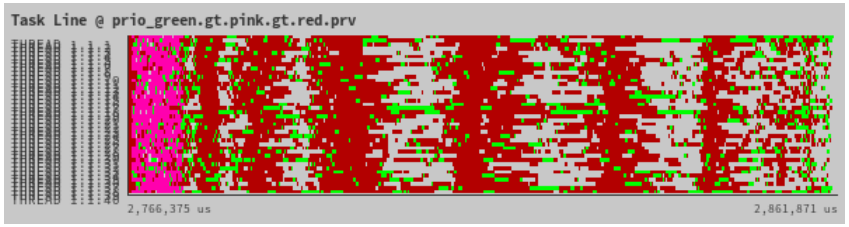
Resulting dependence chains

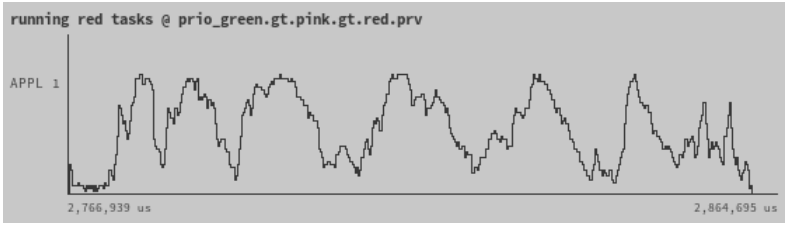


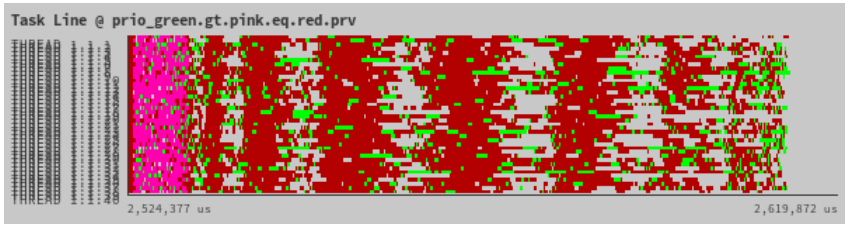
Performance ?

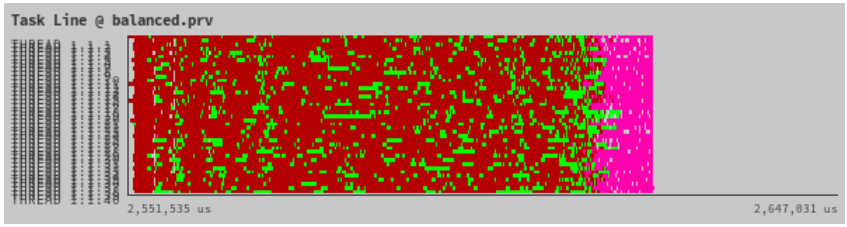
Aqui tendria que poner la correspondiente sin prioridades

- Causes of pulsation of red tasks?
  - Instantiation order and granularity
  - Graph dependencies
- Improvements
  - Priorities
  - Anti-dependence distances
  - Nesting









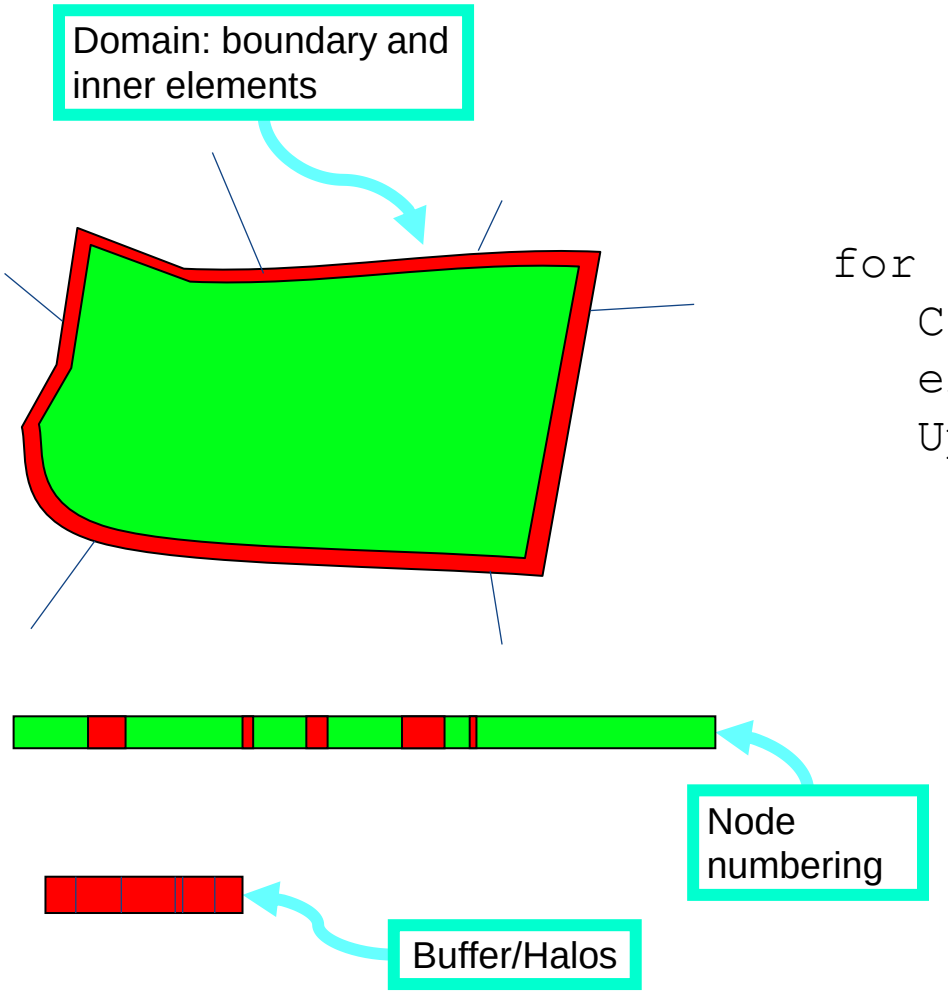
J. Criado et all. “Optimization of Condensed Matter Physics Application with OpenMP Tasking Model”. IWOMP2019

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

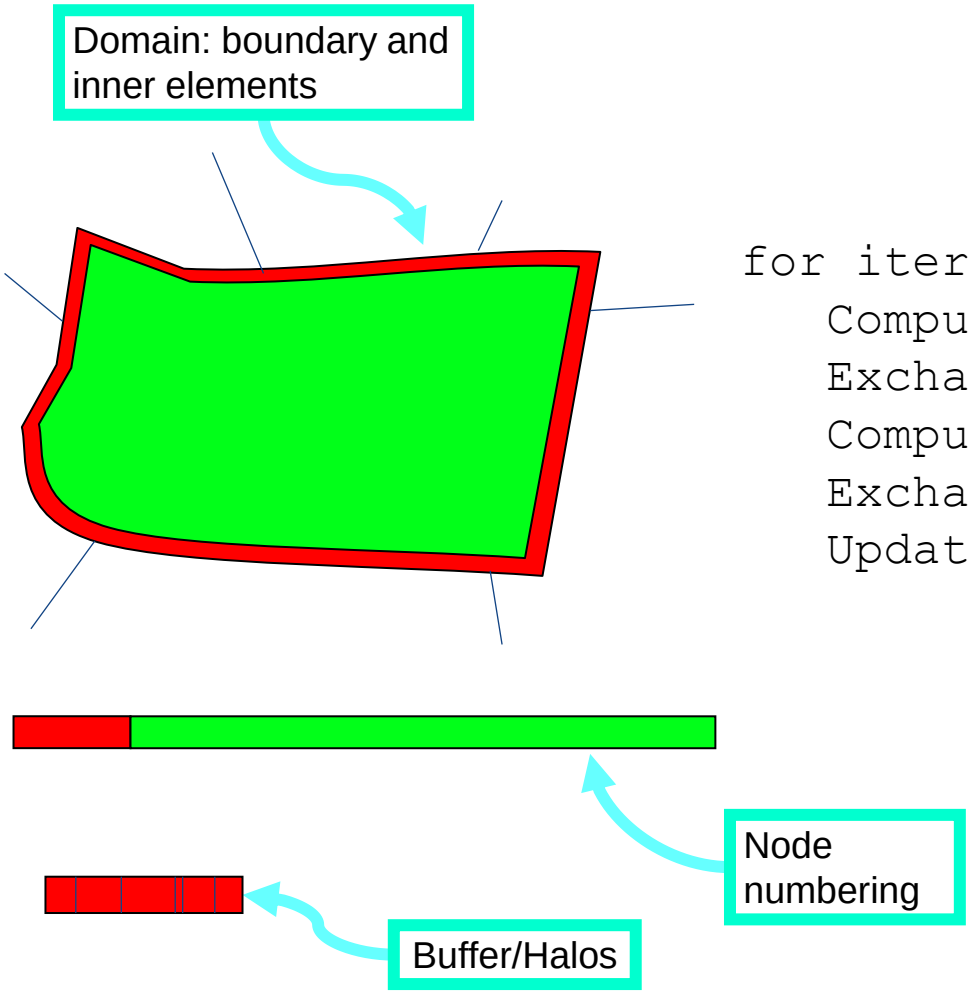
Hierarchical over-decomposition

Non Overlapped MPI code structure



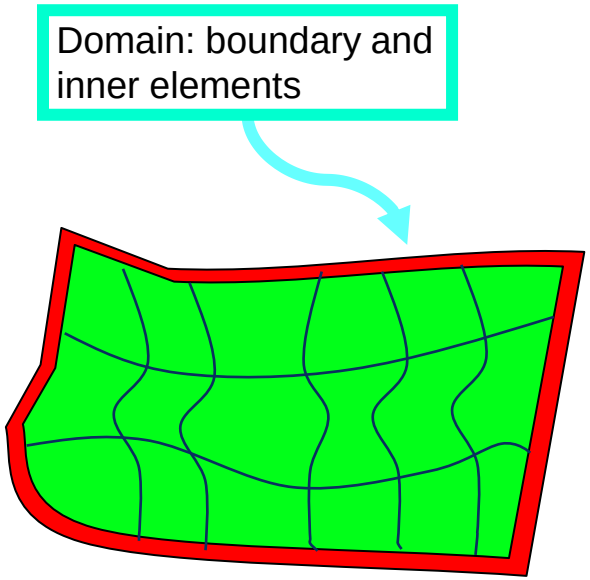

```
for iter
    Compute(red and green)
    exchange(red, neighbors_list)
    Update (red and green)
```

Overlapped MPI code structure

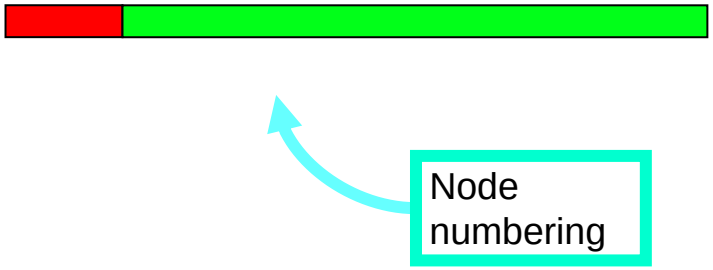



```
for iter
    Compute(red)
    Exchange_issue(red, neighbors_list)
    Compute (Green)
    Exchange_wait
    Update (red and green)
```

Overlapped MPI code structure

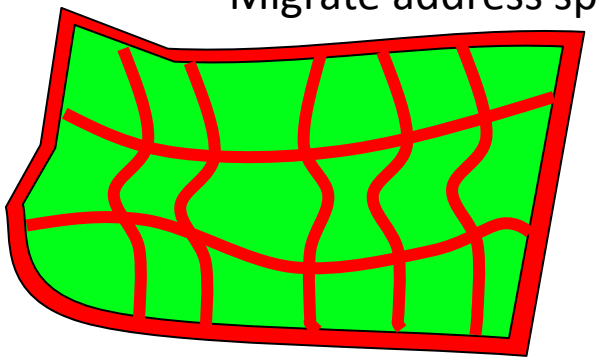


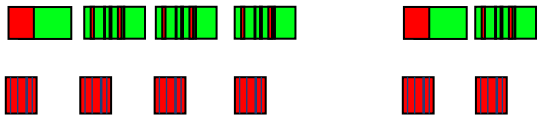

```
for iter
   Compute(red)
   while (green_not_done)
     if (previous_exchange_done)
         issue_iexchange(red_next_neighb)
     Compute (next_green_chunk)
   exchange(red, unfinished neighbours)
   Update (red and green)
```



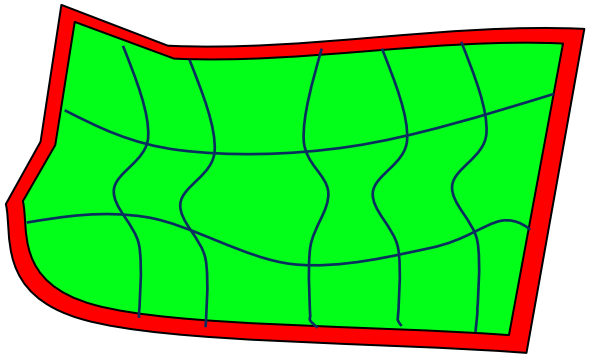
Over-decomposition

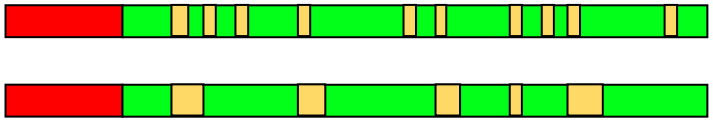
- Processes/objects per core > 1
  - Overlap
  - Load balancing
    - Migrate address space and reroute





Hybrid code structure





```
for iter
   #pragma omp task inout(red)
   Compute(red)
   #pragma omp task inout(red)
   exchange(red, neighbors_list)
   for (Green_blocks)
      #pragma omp task inout(Green_block)
      Compute (Green_block)
   #pragma omp task inout(red, green)
   Update (red and green)
```

+ multideps !!!!

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

revolution

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
      - About tolerating variability by adapting to dynamic situation
    - Performance dominated by overall availability/balance of resources

Closing remarks

- From the latency to the throughput age

- Age before beauty
  - Behavior (insight/models) before syntax
  - Detail performance analytics before aggregated profiles
  - Work instantiation and order before overhead
  - Malleability before fitted rigid structure
  - Possibilities before how tos
  - Elegance before one day shine
  - Power before force

El abuelo cebolleta
ataca de nuevo

The real revolution

- Hybrid Task based ~ OK, but …

- "Proper" model does not guarantee "proper" programs
  - Flexibility → can be used "wrong"
  - Legacy features have a strong "shadow"

- Revolution is in the mindset of programmers
  - "Forget" about hardware, resources
  - Focus on program logic
  - Methodology
    - Top down programming methodology.
      - Every level contributes
    - Throughput oriented
      - try not to stall !
      - First order, then overhead
    - Think global:
      - of potentials rather than how-to's
      - may be unprecise
    - Specify local:
      - needs and outcomes of the functionality being written
      - Precise

Give throughput a try

To exascale ... and before

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Thanks

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Thanks

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

EXCELENCIA SEVERO OCHOA

Thanks

CGPOP @ MPI + OmpSs

```
iter_loop: do m = 1, solv_max_iters
  sumN1=c0
  sumN3=c0
  do i=1,nActive
!omp task in(R) out (Z) inout (sums)
    Z(i) = Minv2(i)*R(i)
    sumN1 = sumN1 + R(i)*Z(i)
    sumN3 = sumN3 + R(i)*R(i)
  enddo
!omp taskwaiton (Z)
  call matvec(n,A,AZ,Z)
  Call MPI_barrier()
  !omp task in(Z,AZ) inout(sum2)
    sumN2=c0
    do i=1,nActive
      sumN2 = sumN2 + AZ(i)*Z(i)
    enddo
  call reduce_update_halo(AZ)
  do i=1,n
    !omp task in(Z) inout(S,X)
      stmp = Z(i) + cg_beta*S(i)
      X(i) = X(i) + cg_alpha*stmp
      S(i) = stmp
  enddo
  do i=1,n
    !omp task in(AZ) inout (Q,R)
      qtmp = AZ(i) + cg_beta*Q(i)
      R(i) = R(i) - cg_alpha*qtmp
      Q(i) = qtmp
  enddo
end do iter_loop
```

B

C

D

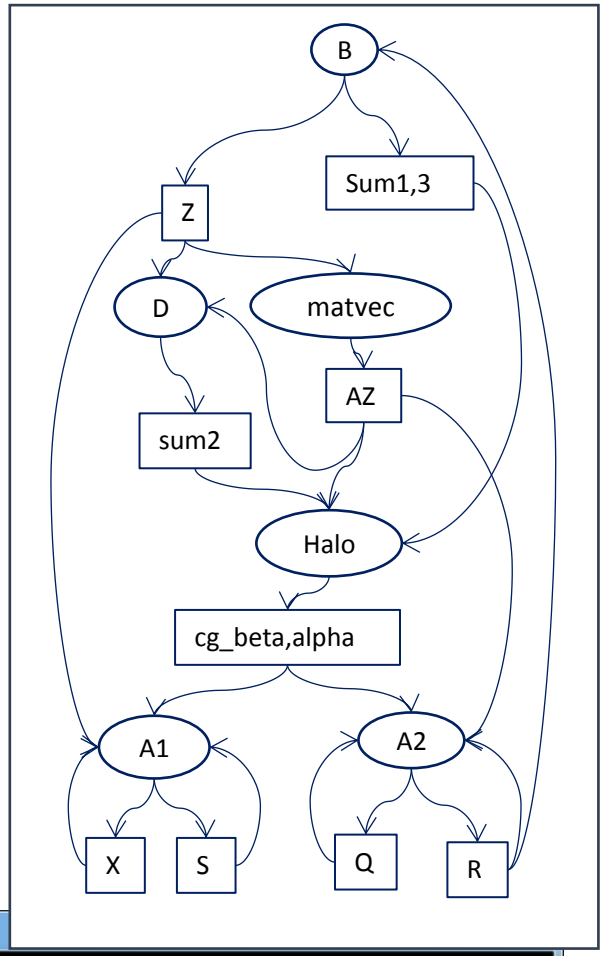
A1

A2

Force DLB

Could be moved before barrier

Not parallelized in this trace (laziness)

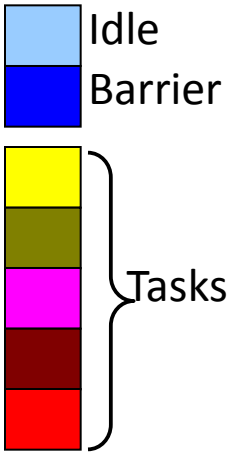


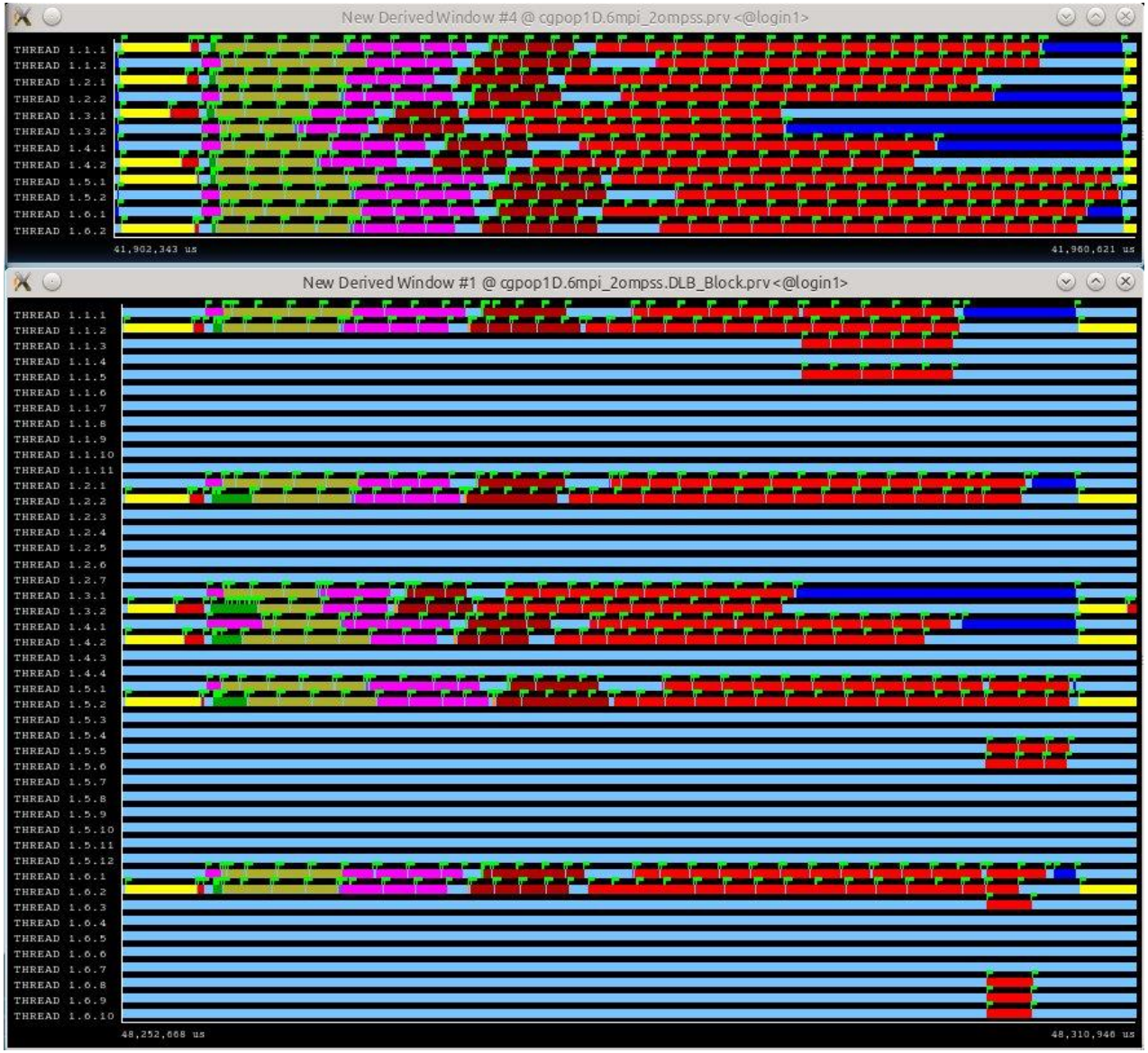
12 MPI x 2 TH

DLB in action: CGPOP

- **Original**
  - 6 MPIs
  - 2 threads x MPI

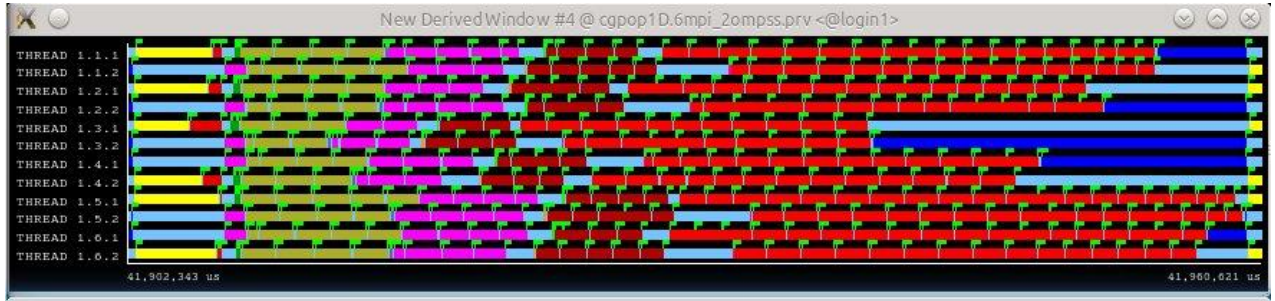
- **DLB**



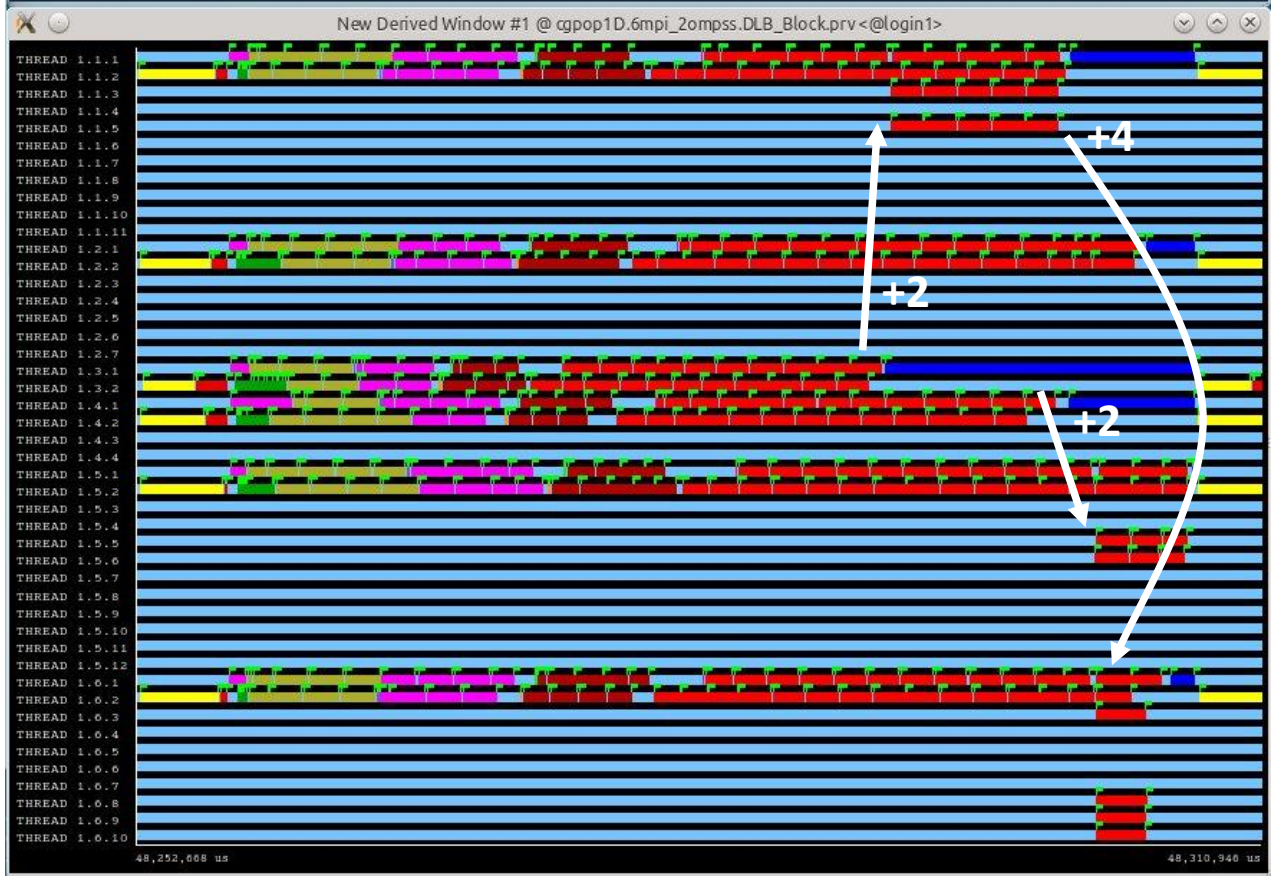


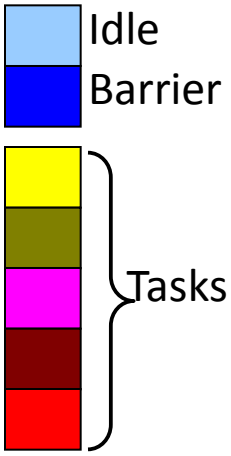
DLB in action: CGPOP

- **Original**
  - 6 MPIs
  - 2 threads x MPI



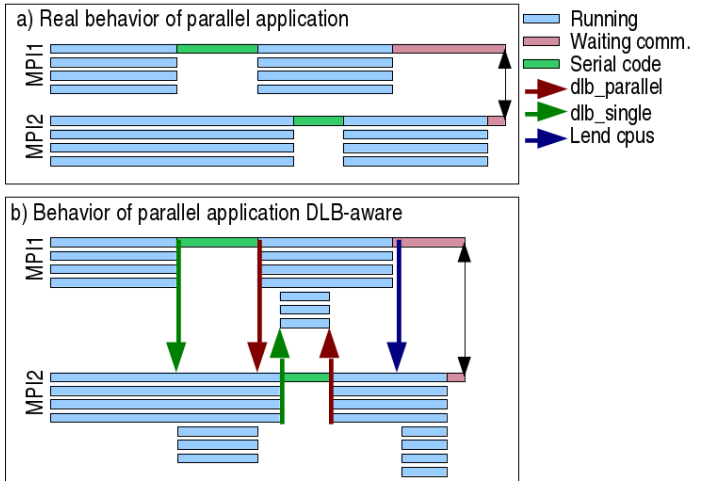
- **DLB**





Exploiting malleability

- Dynamic Load Balance & Resource management
  - Intra/inter process/application
- Library (DLB)
  - Runtime interception (MPIP, OMPT, …)
  - API to hint resource demands
  - Core reallocation policy
- Opportunity to fight Amdalh's law
  - Productive / Easy !!!
    - Nx1
    - Hybridize imbalanced regions

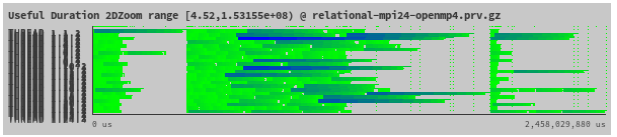


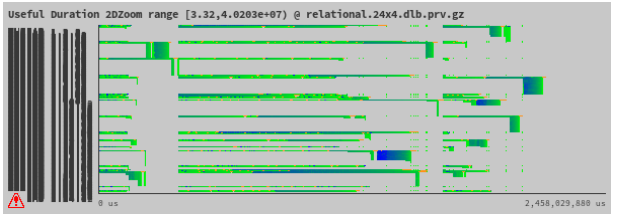
"LeWI: A Runtime Balancing Algorithm for Nested Parallelism". M.Garcia et al. ICPP09
"Hints to improve automatic load balancing with LeWI for hybrid applications" JPDC2014

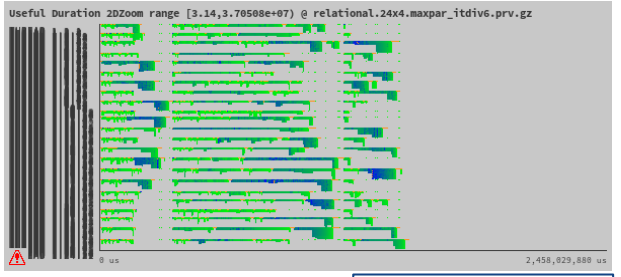
Exploiting malleability

- Dynamic Load Balance & Resource management
  - Intra/inter process/application
- Library (DLB)
  - Runtime interception (MPIP, OMPT, …)
  - API to hint resource demands
  - Core reallocation policy
- Opportunity to fight Amdalh's law
  - Productive / Easy !!!
    - Nx1
    - Hybridize imbalanced regions

dynamic load balancing





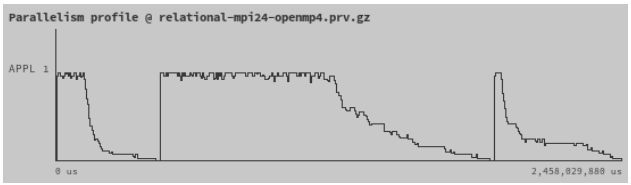


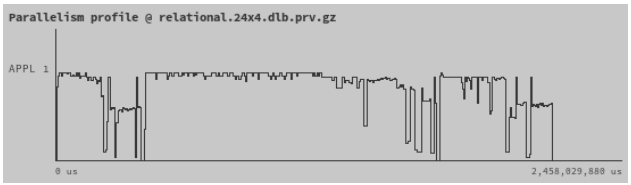
Relational Discovery

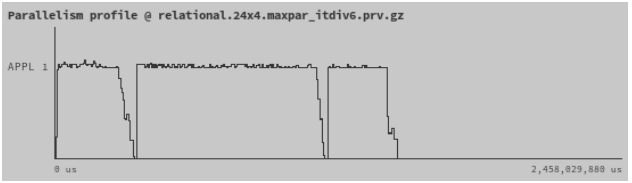
"LeWI: A Runtime Balancing Algorithm for Nested Parallelism". M.Garcia et al. ICPP09
"Hints to improve automatic load balancing with LeWI for hybrid applications" JPDC2014

Exploiting malleability

- Dynamic Load Balance & Resource management
  - Intra/inter process/application
- Library (DLB)
  - Runtime interception (MPIP, OMPT, …)
  - API to hint resource demands
  - Core reallocation policy
- Opportunity to fight Amdalh's law
  - Productive / Easy !!!
    - Nx1
    - Hybridize imbalanced regions





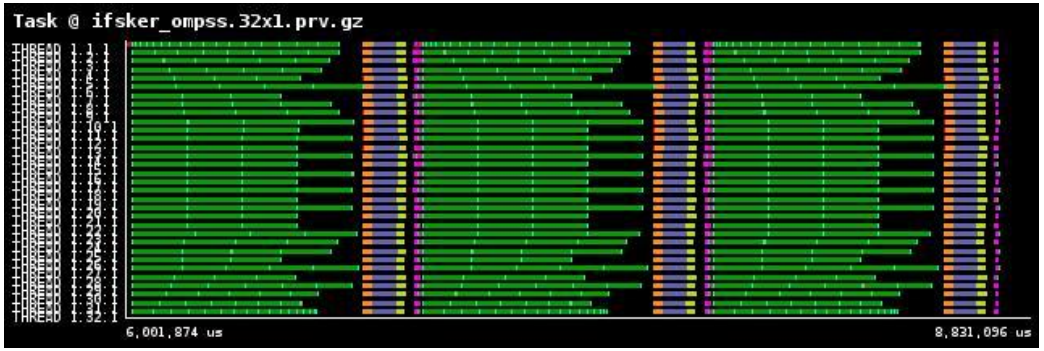


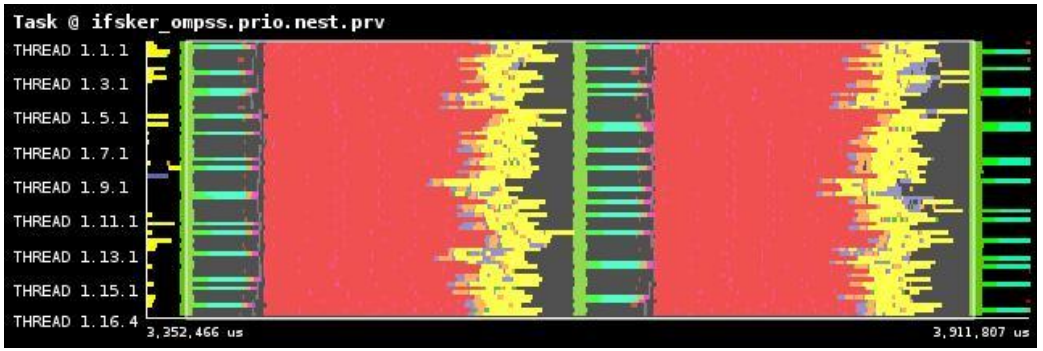
Relational Discovery

"LeWI: A Runtime Balancing Algorithm for Nested Parallelism". M.Garcia et al. ICPP09
"Hints to improve automatic load balancing with LeWI for hybrid applications" JPDC2014

IFS weather code kernel

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





Not same case nor scale
Not unified colors
Just illustrating behavior