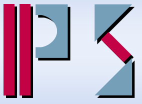


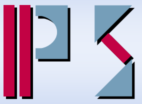
Ubiquitous HPC by Means of Reparallelizable and Migratable OpenMP Applications

Dagstuhl Seminar 07361:
Programming Models for Ubiquitous Parallelism
2007-09-06

Michael Klemm and Michael Philippsen

University of Erlangen-Nuremberg
Computer Science Department 2
(Programming Systems Group)

Why do we need ubiquitous HPC?

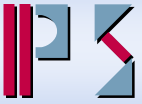


- Researchers often have access to many computing resources around the world.

- However, cluster computing is far from being user-friendly,
  - because of different architectures, interconnects, etc.,
  - and because users have to worry about job schedulers that expect exact estimates of wall clock times for a reservation.
- Thus, most users only exploit a single cluster.

- Estimating the wall clock time is extremely difficult,
  - as runtime depends on the input data and algorithms,
  - and the wall clock time is unpredictably influenced by the cluster's load (e.g. increased network load).

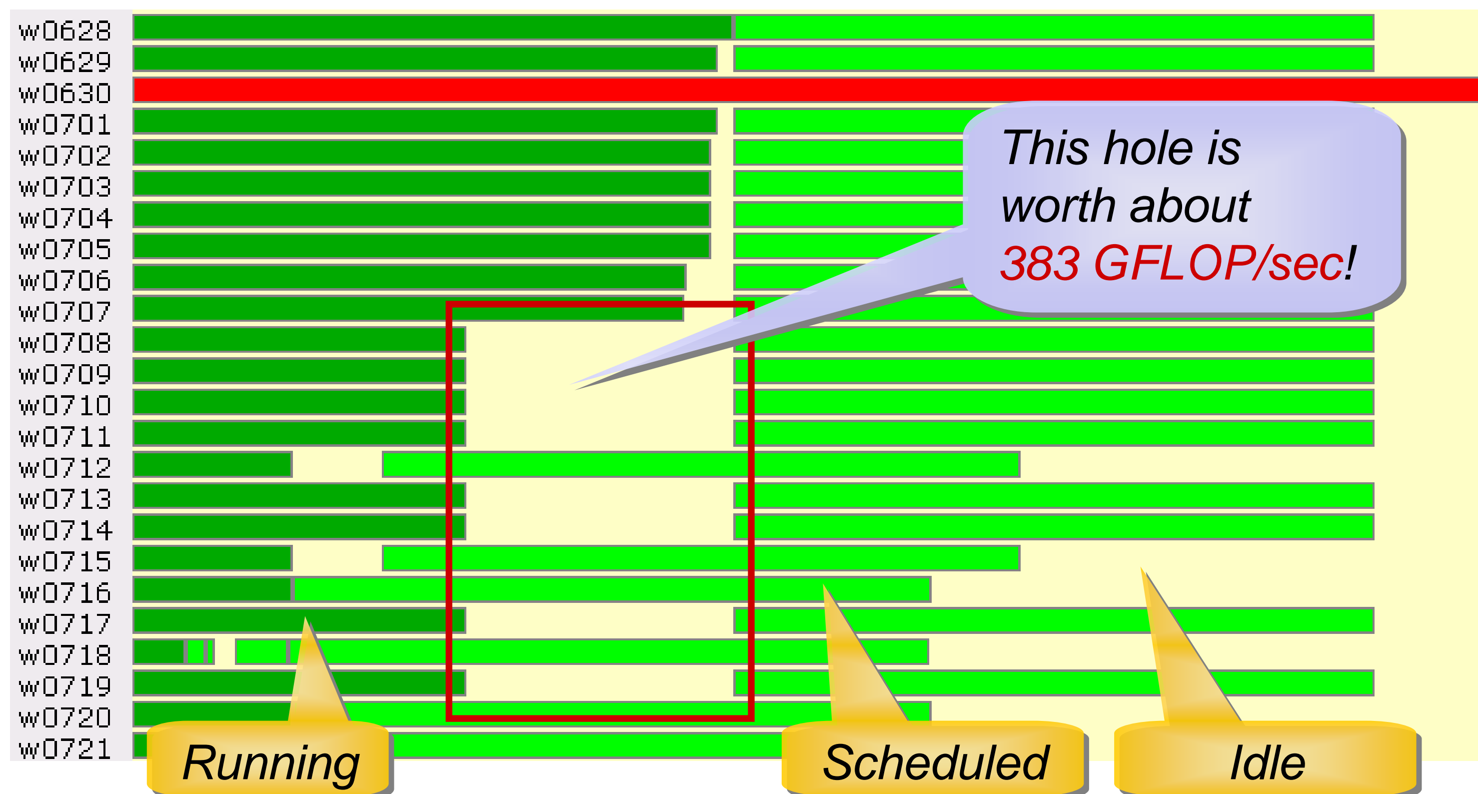
Possible “Solutions”/Remedies



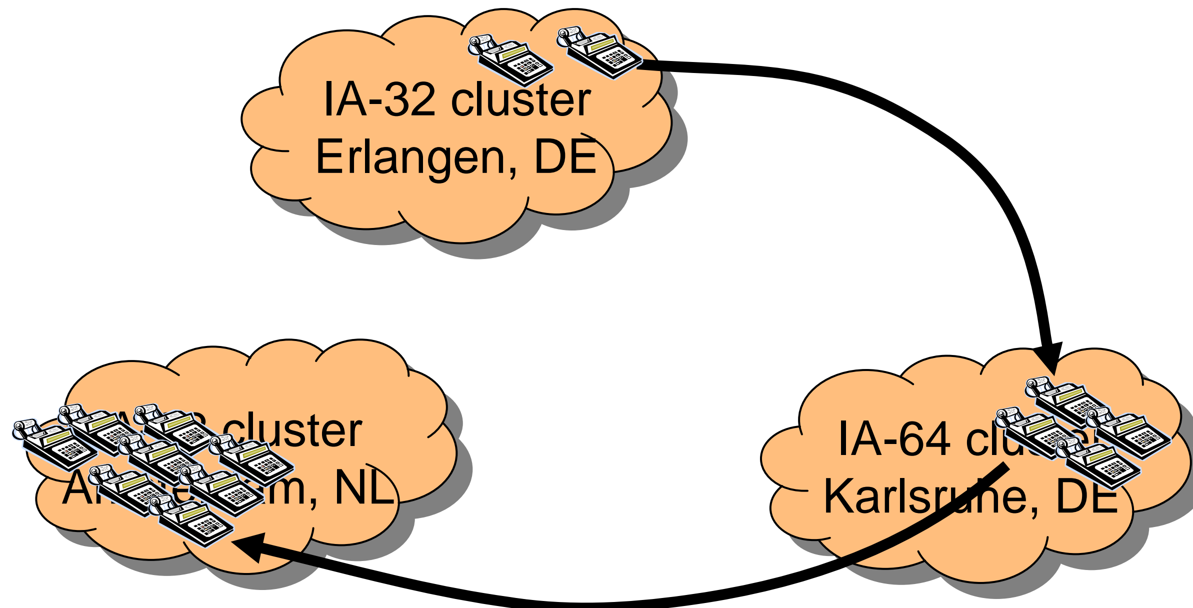
- Decompose the application:
  - Small, independent phases of execution
  - Save intermediate results after each phase
  - Restoration becomes possible after each phase
  - Drawbacks:
    - Additional programming effort
    - Complexity of the application grows
- Overestimate runtime:
  - Often the runtime estimate is doubled by users
  - Or: request maximum walltime allowed by the cluster’s queues
  - Drawbacks:
    - Long waiting times until actual job execution
    - Scheduler suffers from reservation holes

Reservation Holes

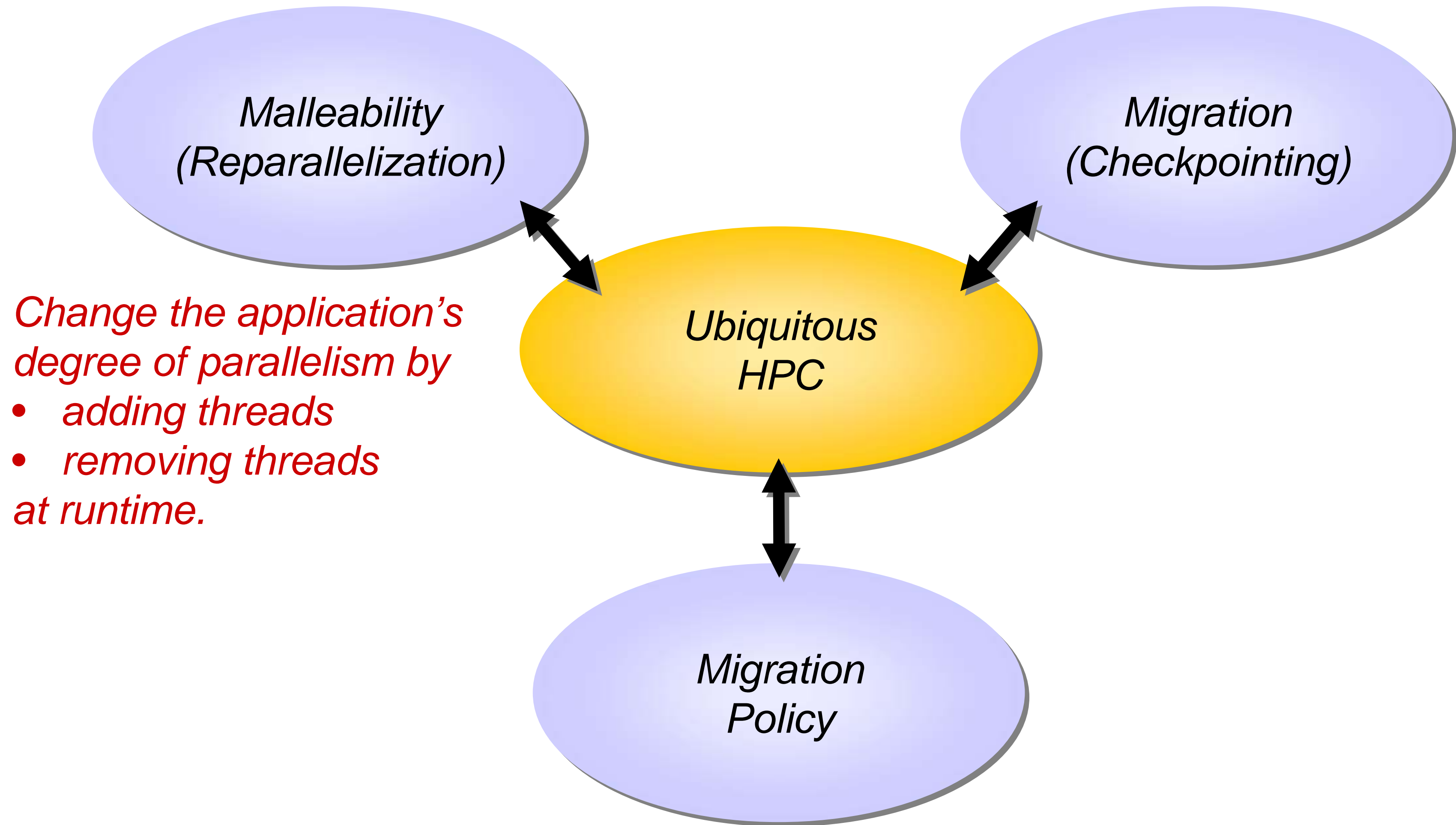
- Clusters have many idle nodes due to over-estimated execution times:
  - Waste of energy by running and cooling the idle nodes
  - Waste of money due to administrative costs, hardware expenses



- Possible user-friendly solution:
  - User starts the job at any point in the Grid (with an arbitrary walltime and CPU count)
  - Application independently requests a new reservation after reaching the end of the time slice
  - Migration provides a means to access free resources in the Grid

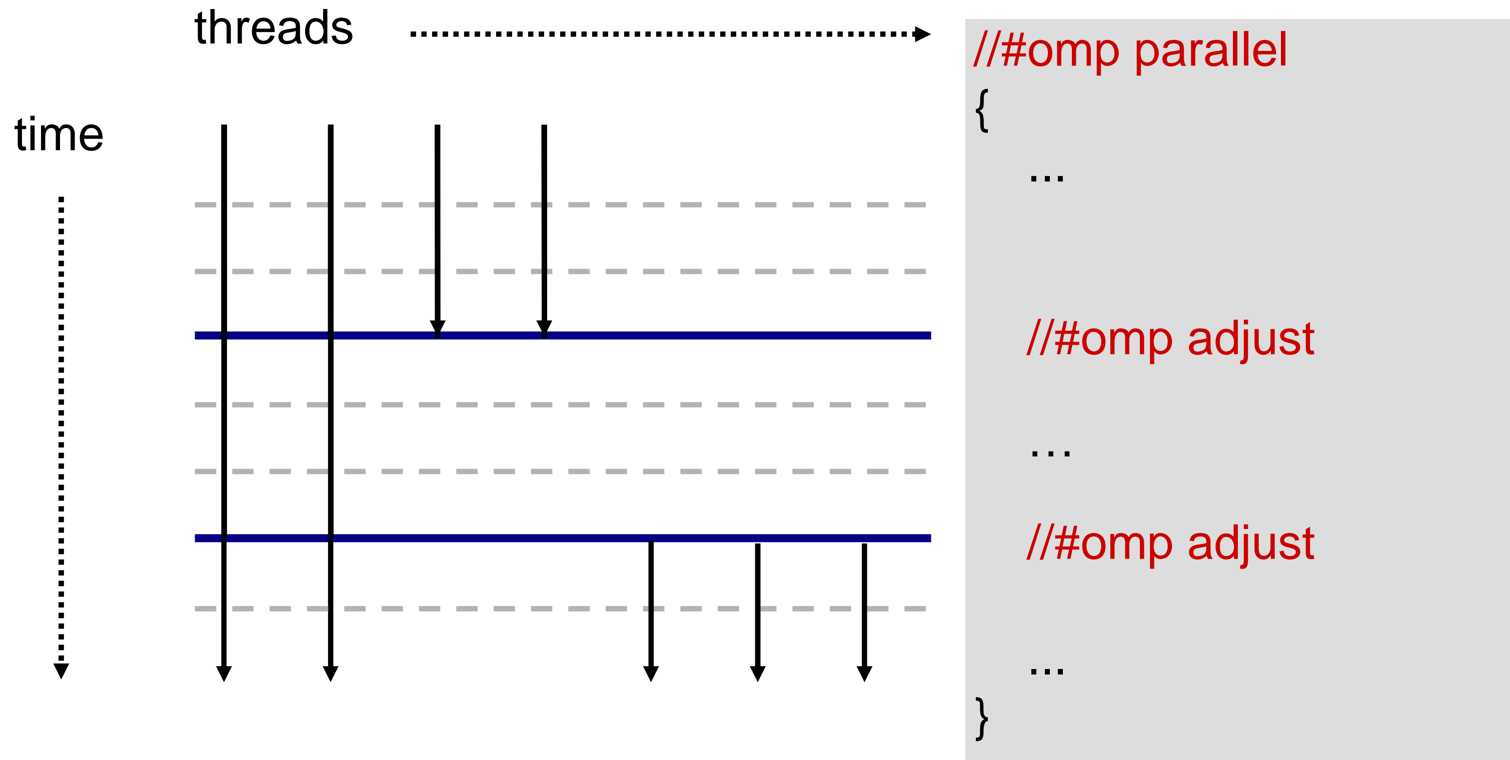


Building Blocks



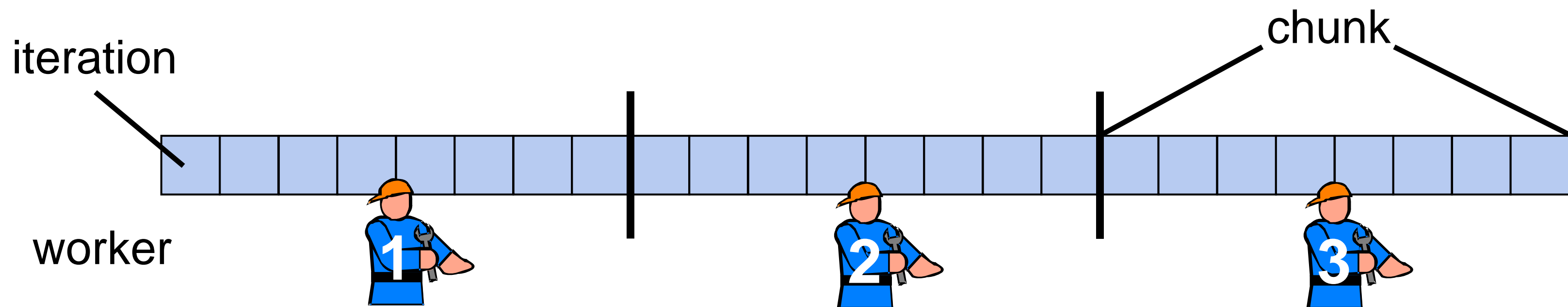
Reparallelization

- Reparallelization only occurs at adjustment points:



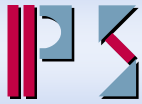
Reparallelization of Statically Distributed Loops

- Example partitioning of the iteration space:
  - Static: iteration space is evenly distributed among the workers

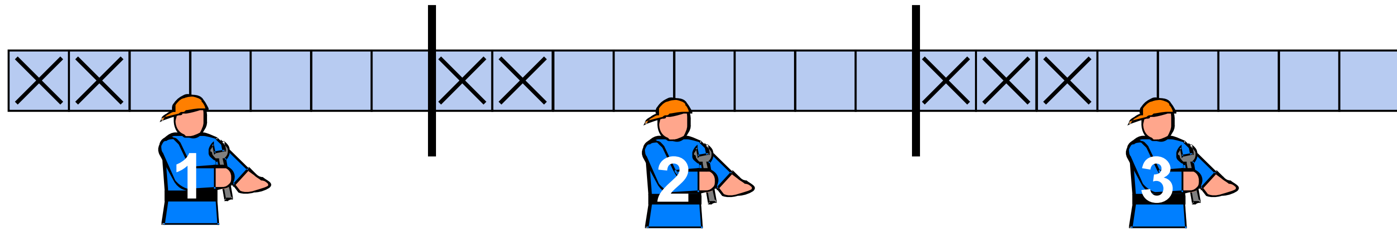


- Reparallelization implies redistributing the work initially assigned to the different worker threads:
  - Already processed iterations must not be recomputed, otherwise loop semantics are violated
  - Mark finished iterations in a bitlist

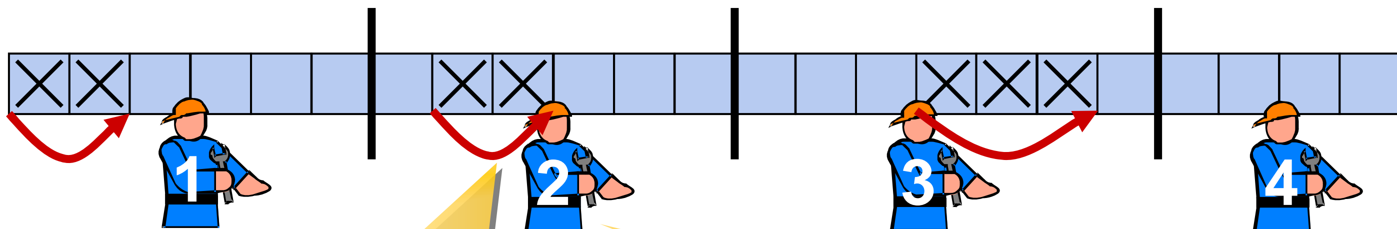
Reparallelization of Statically Distributed Loops – 2



- Iteration space before redistribution:



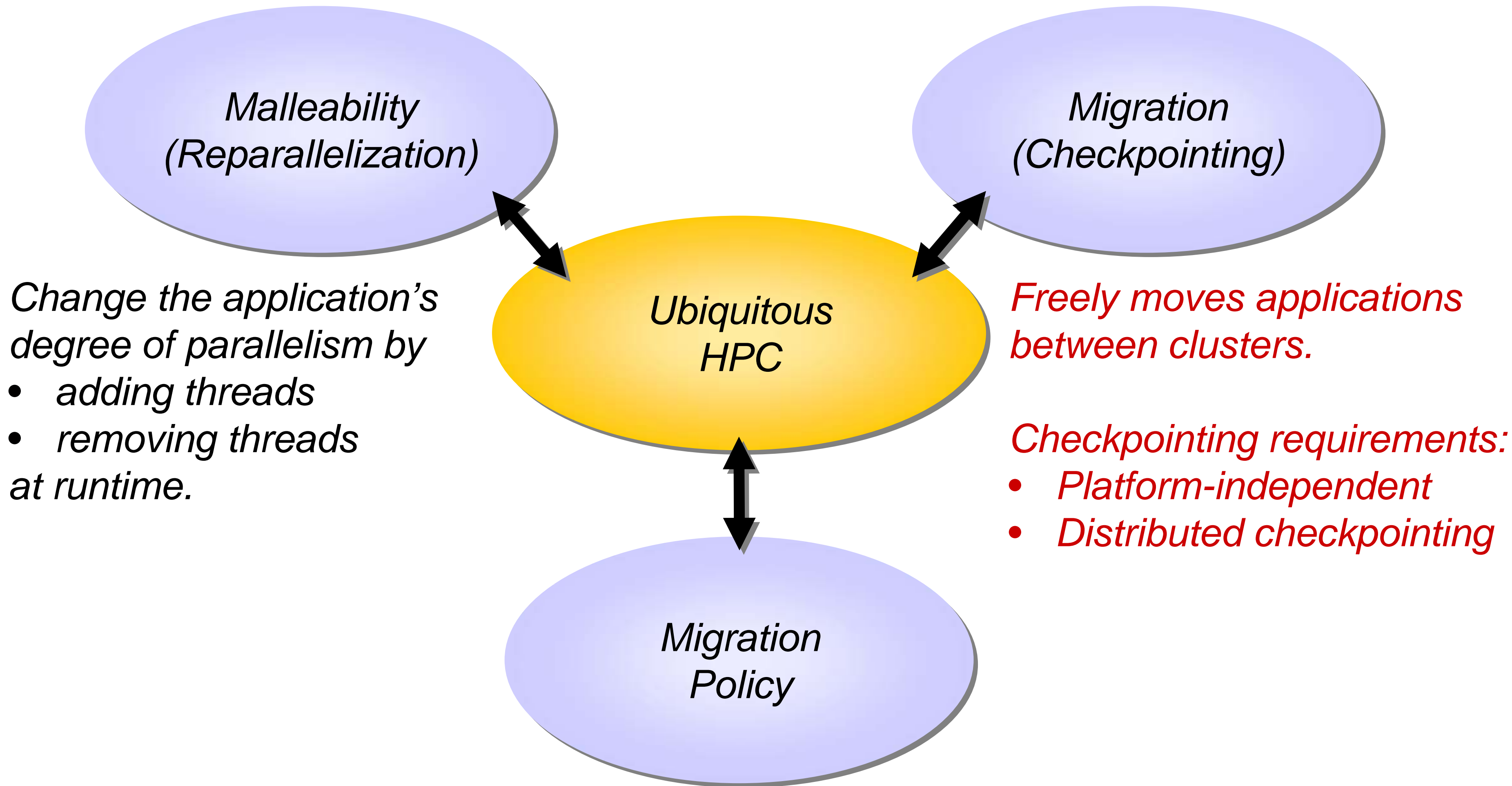
- Iteration space after redistribution:



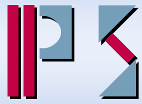
2. *Already computed iterations are skipped.*

1. *Every worker receives the same number of unprocessed iterations.*

Building Blocks

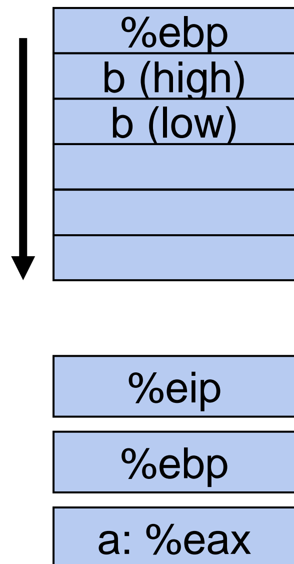


Platform-independent Checkpointing



- On each architecture the stack looks different

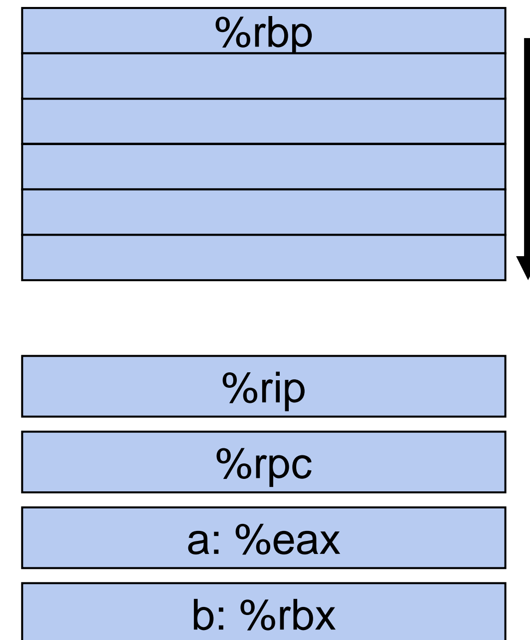
Stack and registers (IA-32)



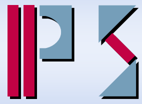

```
long checkpointee(int c) {
    int a = 0;
    long b = 1000;
    if (c != 0)
        a = a + 1;
    checkpoint();
    return a + b;
}
```

live: *{a,b}*
dead: *{c}*

Stack and registers (x86-64)

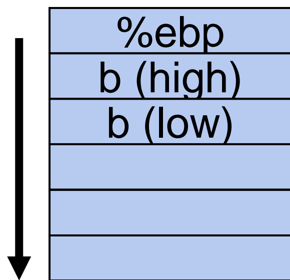


Platform-independent Checkpointing – 2



- On each architecture the stack looks different
- Store a thread's stack in a platform-independent stack format
- Compiler creates functions for the stack mapping

Stack and registers (IA-32)



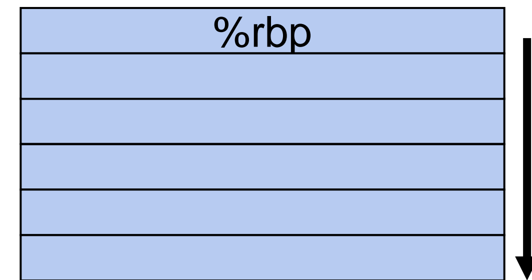
Stack and registers (generic)

checkpointee

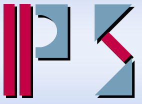
a: "+ C:0@B0 C:1@B1"
b: "C:1000@B:0"

Unified Descriptor String
for each live variable
(compiler generated).

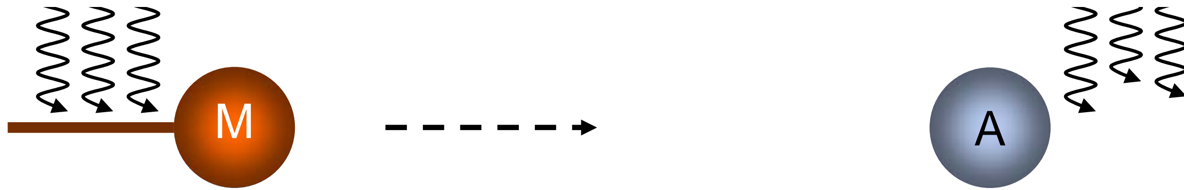
Stack and registers (x86-64)



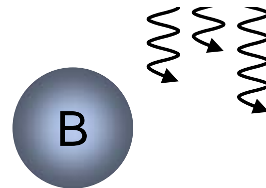
Platform-independent Checkpointing – 3a



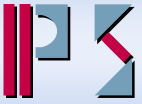
- Checkpointing of applications:
  - Empty the network
  - Checkpoint local threads
  - Serialize all reachable objects on the heap



- - -▸ data message

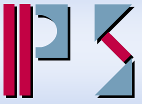


Platform-independent Checkpointing – 3b



- Checkpointing of applications
  - Empty the network
  - Checkpoint local threads
  - Serialize all reachable objects on the heap

Platform-independent Checkpointing – 3c



- Checkpointing of applications
  - Empty the network
  - Checkpoint local threads
  - Serialize all reachable objects on the heap

M

A

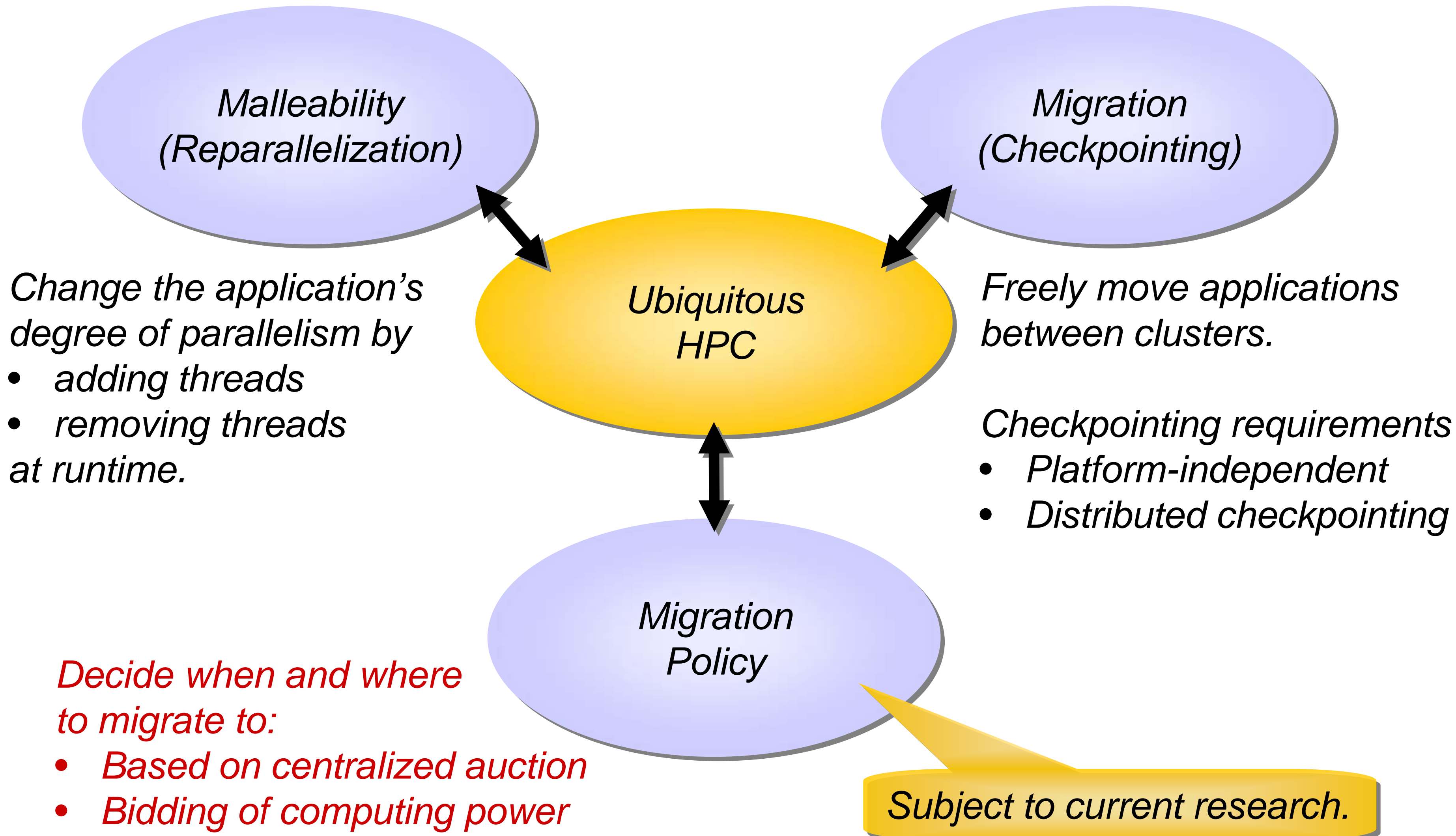
B

- - - -▸ data message

⟶ control message

- *$O(n^2)$* control messages
- Duration: 10ms to 20ms (16 nodes)

Building Blocks



Centralized Auctions

- Clusters periodically report a bid to the application:
  - Number of CPUs and nodes
  - CPU speed
  - etc.

IA-32 cluster
Erlangen, DE

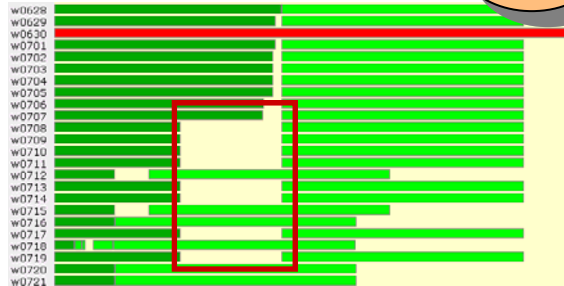
In 24 hours 8 CPUs with
41.6 GFLOP/sec for 48 hours

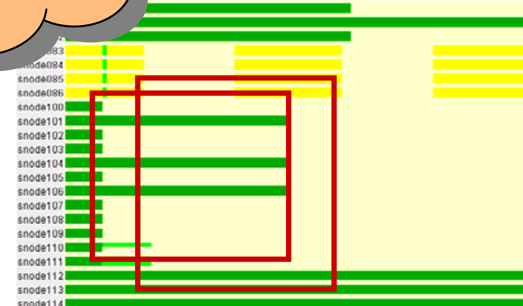
In 66 hours 32 CPUs with
383 GFLOP/sec for
24 hours

In 48 hours 9 CPUs with
46.8 GFLOP/sec for
48 hours

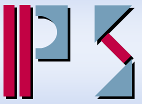
IA-32 cluster
Amsterdam, NL

IA-64 cluster
Karlsruhe, DE

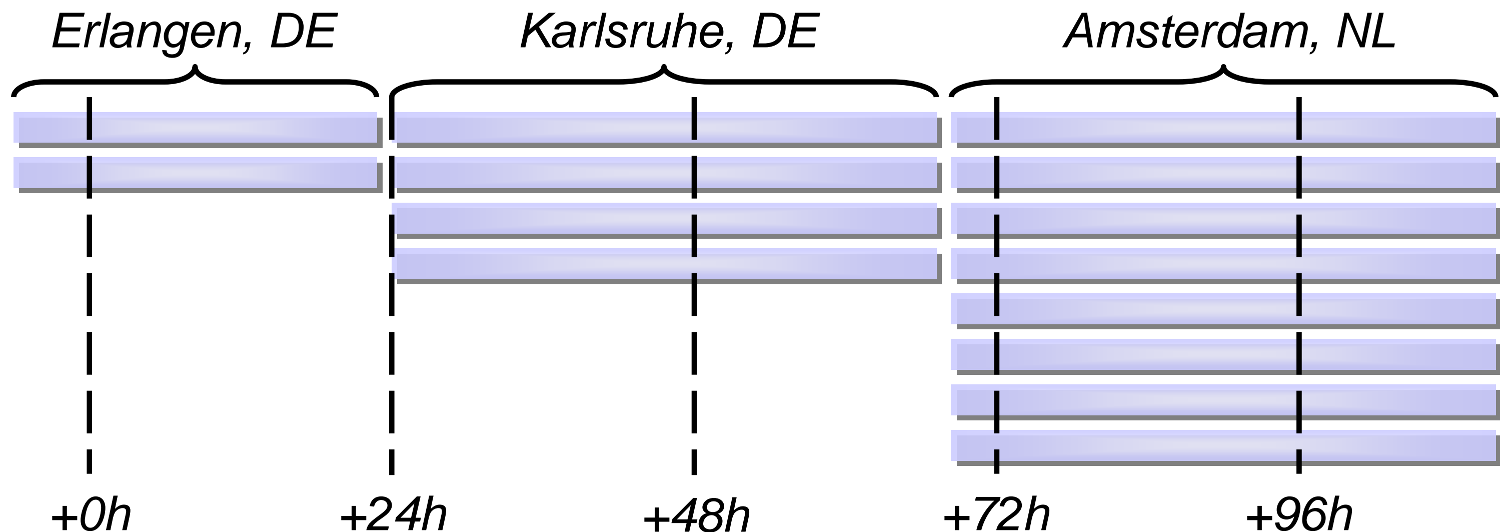




Centralized Auctions – 2

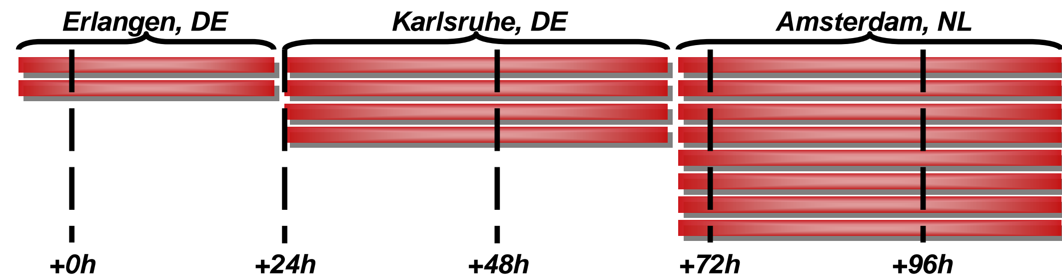


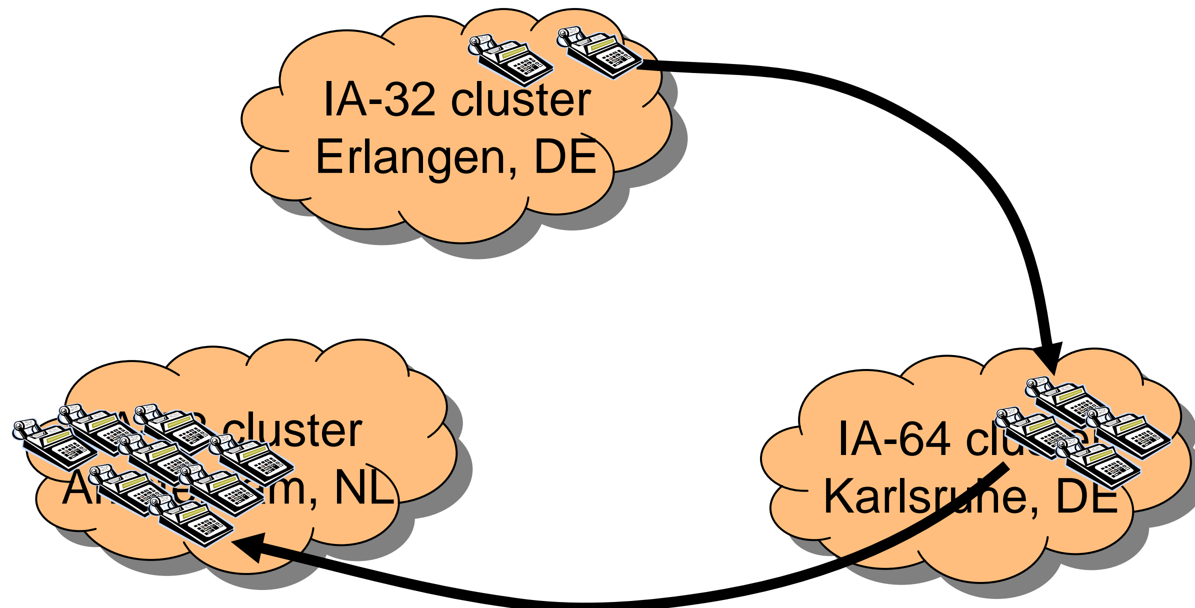
- The application's runtime system creates a tentative schedule out of the incoming bids:
  - maximize the CPU power consumed
  - minimize the number of expensive migrations
- If the situation at the clusters changes, the schedule is updated



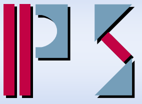
Centralized Auctions – 3

- According to the schedule, the application migrates to a new target, by
  - checkpointing its state and data
  - adapting to the new CPU count

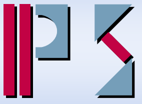




Summary



- Grid Computing is far from being as easy as desktop computing which we all are used to
- Migratable applications make Grid computing easier:
  - Much easier job submission for the user
  - Boundaries between clusters are blurred
- Schedulers do not suffer from reservation holes:
  - Adjust the application to the number of free nodes in the cluster
  - Approximate the application's runtime piece by piece
- Enable fair-share scheduling for the machine (one of the hot topics in current cluster research)



Thank you for your attention!

Any questions?