Satin: a High-Level and Efficient Grid Programming Model

Aurél Czövek
czaurel@gmail.com

Vrije Universiteit Amsterdam - Faculty of Sciences

February 24, 2012
Figure: Not this one.
Common issues of grids

- WAN connections are slow.
- Resources in the grid are heterogenous (hardware, OS, ...).
- Grids are dynamic. Failures are more common than in clusters, and resources come and go.
- Managing these is hard and error prone.
Common issues of grids

- WAN connections are slow.
- Resources in the grid are heterogeneous (hardware, OS, ...).
- Grids are dynamic. Failures are more common than in clusters, and resources come and go.
- Managing these is hard and error prone.

Result: Grids are used for Master-Worker apps (or just as a shared batch queue).
Aims of Satin

Define a high-level grid programming model with the ability to:

- Hide complexity of grids from the programmer.
- Manage grid-related issues automatically.
- Run efficiently on grids.
- Support a reasonably broad range of applications.
Satin exploits the hierarchical nature of grids by using an *Extended Divide & Conquer* model (*Master-Worker* is 'special case').

Recursively split subcomputations, using primitives:

- *Spawn* an asynchronous job, which implements the *satin.Spawnable* interface.
- Wait for results with *sync*.

For more general usability, extended by:

- Shared objects
- Speculative parallelism
Divide & Conquer on a grid

cluster1

job1

cluster2

job2

job4

job5

cluster3

job3

job6

job7

cluster4
Basic implementation

Java: "Write once, run everywhere".

Figure: Compiling and running Satin code.
On spawnable method call, Satin creates an invocation record (method, parameters, handle for return value).

The record is inserted in a work queue (1 / JVM). \texttt{sync} executes methods from here.

Job movement between queues is done by the load-balancing algorithm.

Max. 1 / 150 jobs is transferred to a remote machine: \textit{serialization on demand}
Invocation records are stored at the head of a double-ended job queue. sync starts also from the head of the queue.

Idle nodes start stealing work from the tail of other nodes’ queues. In this way, large-grain jobs (higher up the divide & conquer tree) are stolen, reducing communication overhead.

Random stealing is known to achieve optimal load-balancing for homogenous systems (single cluster).
Load-balancing

- Invocation records are stored at the head of a double-ended job queue. `sync` starts also from the head of the queue.
- Idle nodes start stealing work from the tail of other nodes’ queues. In this way, large-grain jobs (higher up the divide & conquer tree) are stolen, reducing communication overhead.
- Random stealing is known to achieve optimal load-balancing for homogenous systems (single cluster).

However, for $C$ clusters, $(C - 1)/C \times 100\%$ of all steal requests will go to remote clusters.
Idle nodes try to steal from a node in a remote cluster (async).
Until reply arrives, avoid stealing other remove jobs by setting a flag.
While waiting, continue with RS from local nodes (sync).
Combines advantages of RS with limited WAN communication
void cluster_aware_random_stealing(void){
    while(NOT exiting) {
        job = queue_get_from_head();
        if(job) {
            execute(job);
        } else {
            if(nr_clusters > 1 AND NOT stealing_remotely) {
                /* no wide-area message in transit */
                stealing_remotely = true;
                send_async_steal_request(remote_victim());
            }
            /* do a synchronous steal in my cluster */
            job = send_steal_request(local_victim());
            if(job) queue_add_to_tail(job);
        }
    }
}

void handle_wide_area_reply(Job job){
    if(job) queue_add_to_tail(job);
    stealing_remotely = false;
}
Load-balancing speedup

Figure: RS and CRS speedup for various test applications (2001).
Further features

Satin also implements other features, all of them transparently:

- Malleability and fault tolerance, by reusing the work of *orphan tasks* and periodically *checkpointing* the system
- Self-adaptivity, enabling automatic addition/removal of nodes during execution
Further features

Satin also implements other features, all of them transparently:

- Malleability and fault tolerance, by reusing the work of *orphan tasks* and periodically *checkpointing* the system
- Self-adaptivity, enabling automatic addition/removal of nodes during execution

Since the programming model has no concept of machines, the programmer can **not** directly influence these.
Adaptivity

- Separate coordinator process periodically estimates the resource requirements of the application.
- Efficiency is estimated by the *weighted average efficiency* formula:

\[
wa\text{-}efficiency = \frac{1}{n} \sum_{i=1}^{n} speed_i \times (1 - overhead_i)
\]

where \( n \) is the number of processors and \( overhead_i \) is the fraction of time the \( i^{th} \) processor spends being idle or communicating. The speed is periodically estimated by running a benchmark.
The coordinator tries to keep value between $E_{min}$ and $E_{max}$

- Above $E_{max}$ it requests new processors (used 0.5)
- Below $E_{min}$ it starts removing them (used 0.3)

The coordinator also computes inter-cluster overhead and can remove entire clusters if needed
Adaptivity speedup

**Figure:** Speedup with adaptivity in different scenarios. First one is overhead test.
Conclusion

- High-level, transparent grid programming model
- Cluster-aware Random Stealing enables efficient job distribution
- Considerable performance gain by adding/removing resources automatically
- Many more features (read paper!)
Any questions?