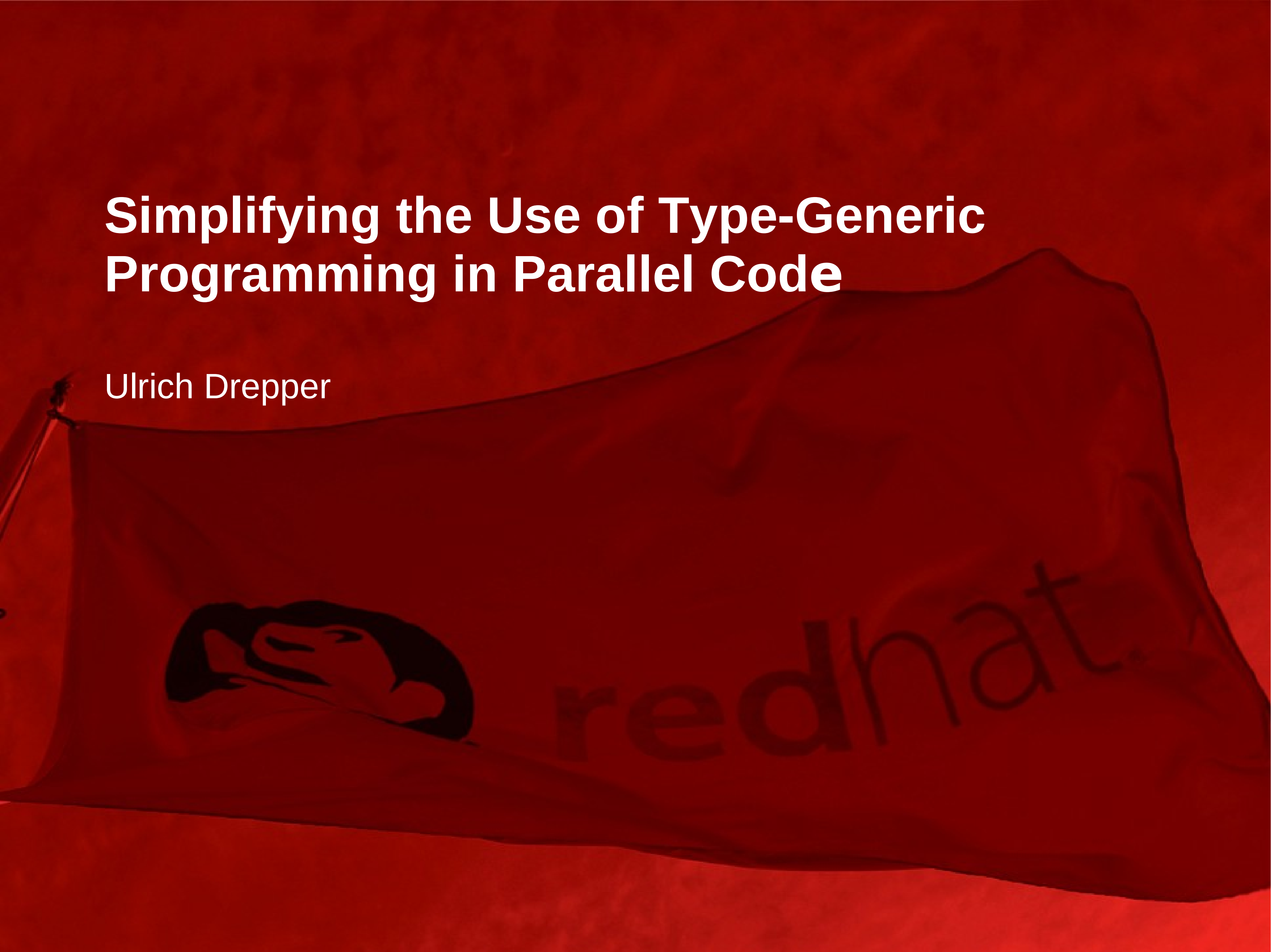
Simplifying the Use of Type-Generic Programming in Parallel Code

Ulrich Drepper



Commercial Programming

- In *my* world:
  - Java (unfortunately) for web and business logic
  - C++ for everything else
    - Especially also when performance is an issue
  - C++ as in ISO C++ (1998 and now 201x)
    - *Not* OOP!!! (People can learn from mistakes)
    - Type generic programming
      - ISO C++ 201x will allow most of the TG Programming theory to be applied

Type-Generic Programming & C++

- Now (ISO C++ 201x) good language support
- A lot of library support
  - Containers, algorithms
  - Combined with functional programming aspects (lambdas)
- Language even includes support for thread handling

- But: no integration of parallel programming into the library
  - No thread-safety guarantees
  - No explicit support for thread-safety
  - Not easy/possible to integrate in existing APIs

C++ map Class

- Type-Generic class in C++:

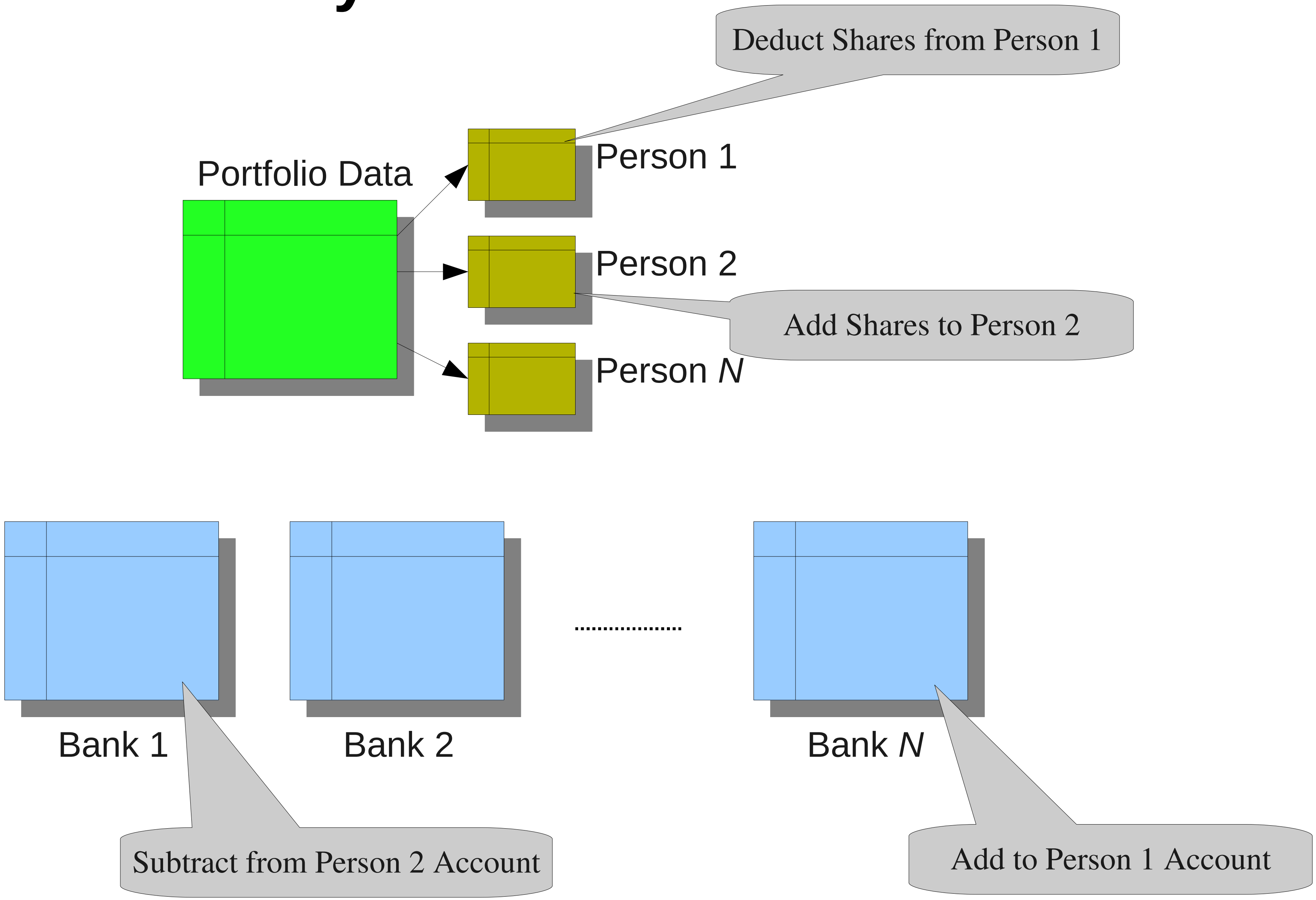
```
template <class Key, class T, class Compare=less<Key>,
          class Allocator=allocator<pair<const Key, T> > >
class map
```

- All type parameters
- References to global objects only alternative
  - Unpractical for almost all uses
  - Need to know ahead of time how many mutexes

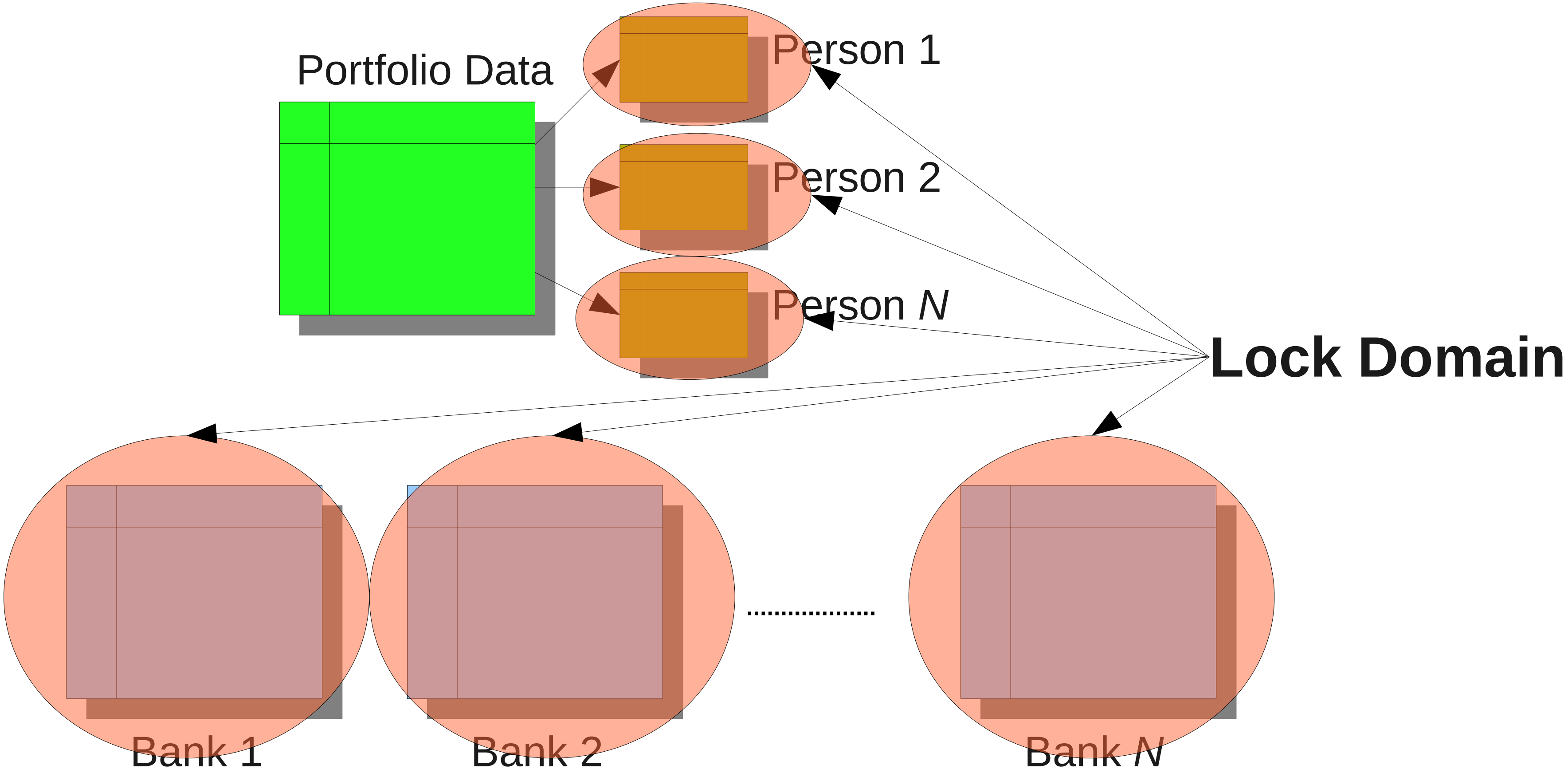
This leaves us with...

- Explicit, external locking
- With all the associated problems:
  - Selection of granularity
  - Error-prone use
    - Forget to use
    - AB-BA deadlocks

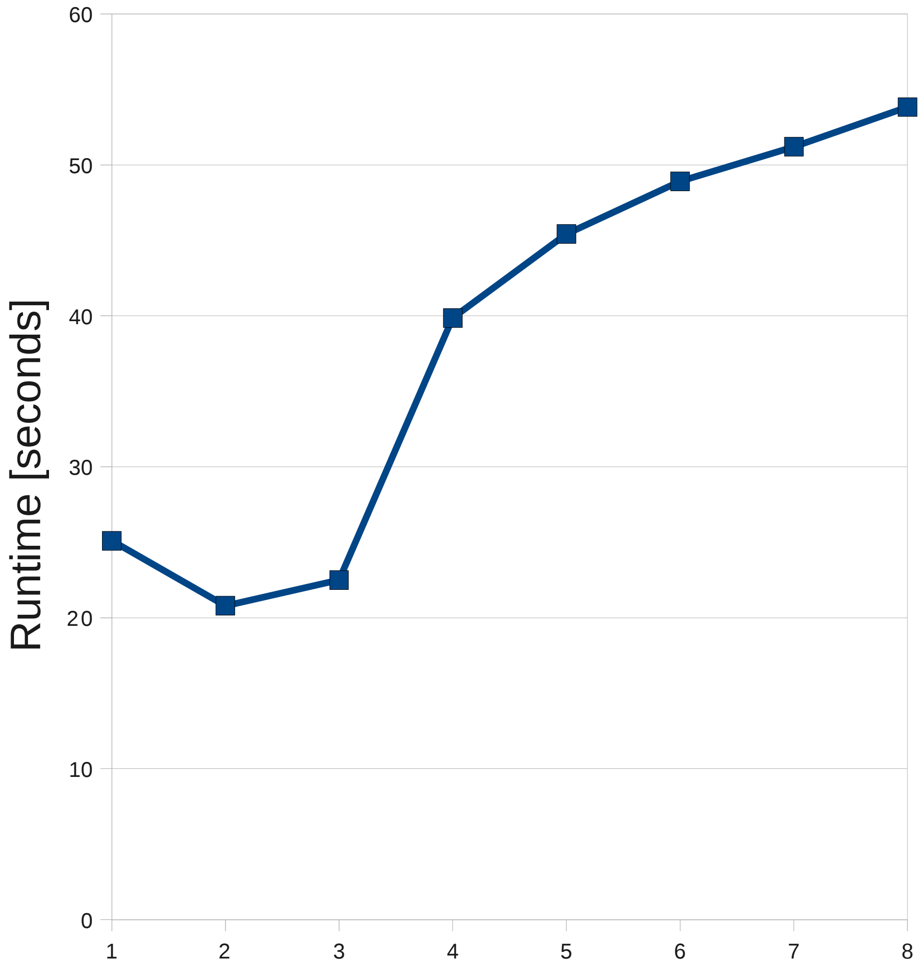
Transaction System



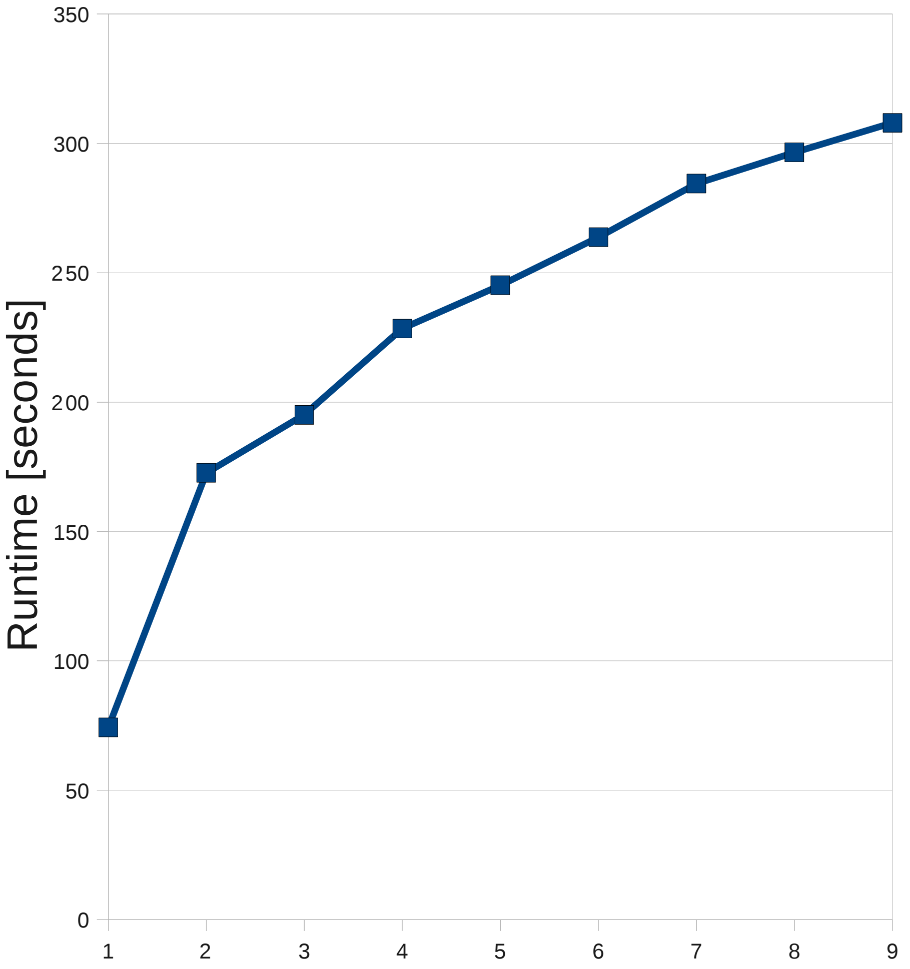
Trying To Parallelize



Not What We Want



Single Core i7



Opteron NUMA

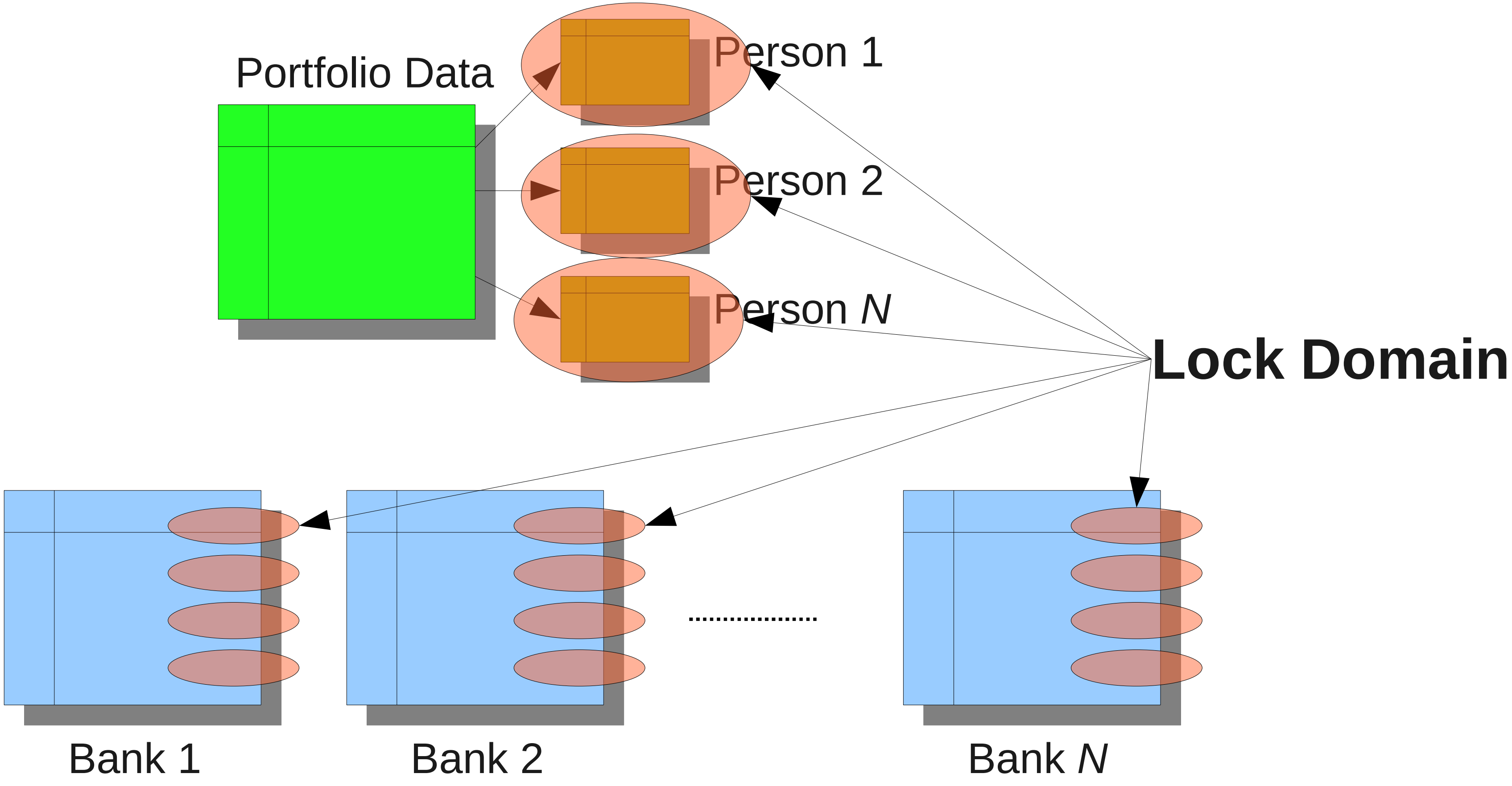
Too Little Parallelism

- Idealized Amdahl's Law

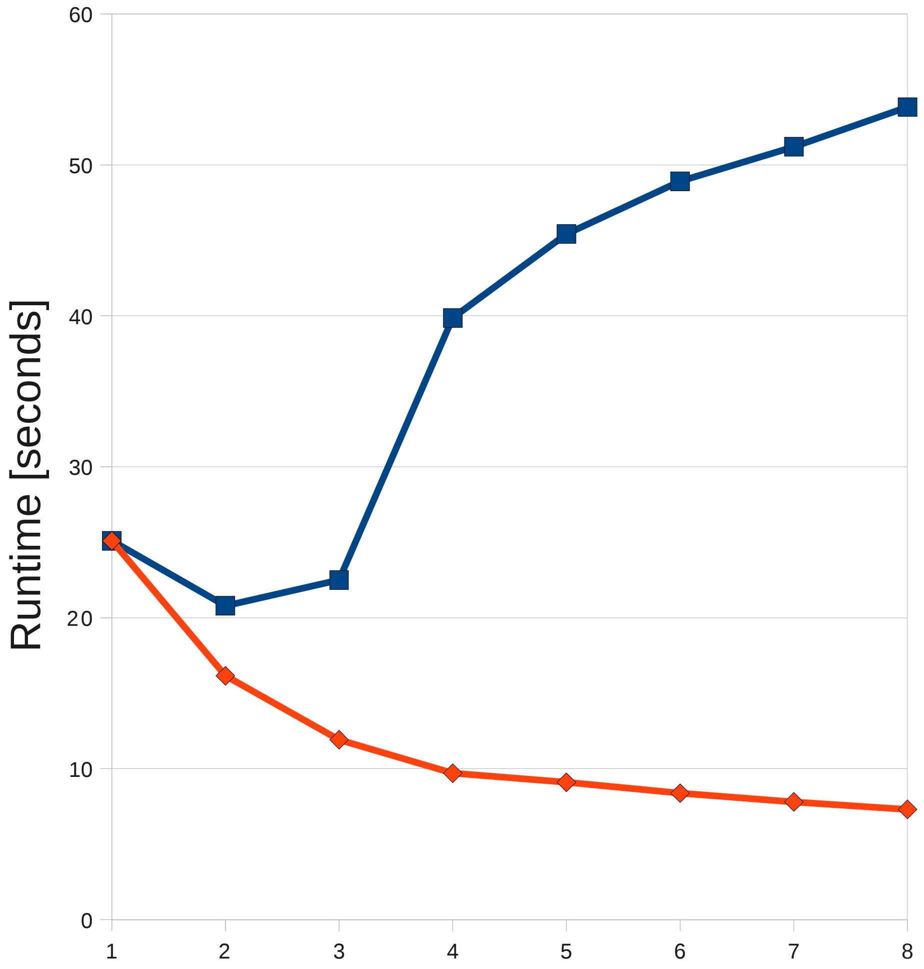
$$S = \frac{1}{(1-\boldsymbol{P}) + \frac{\boldsymbol{P}}{N}}$$

- *P* is too small
- After lock contention analysis: push locks further down

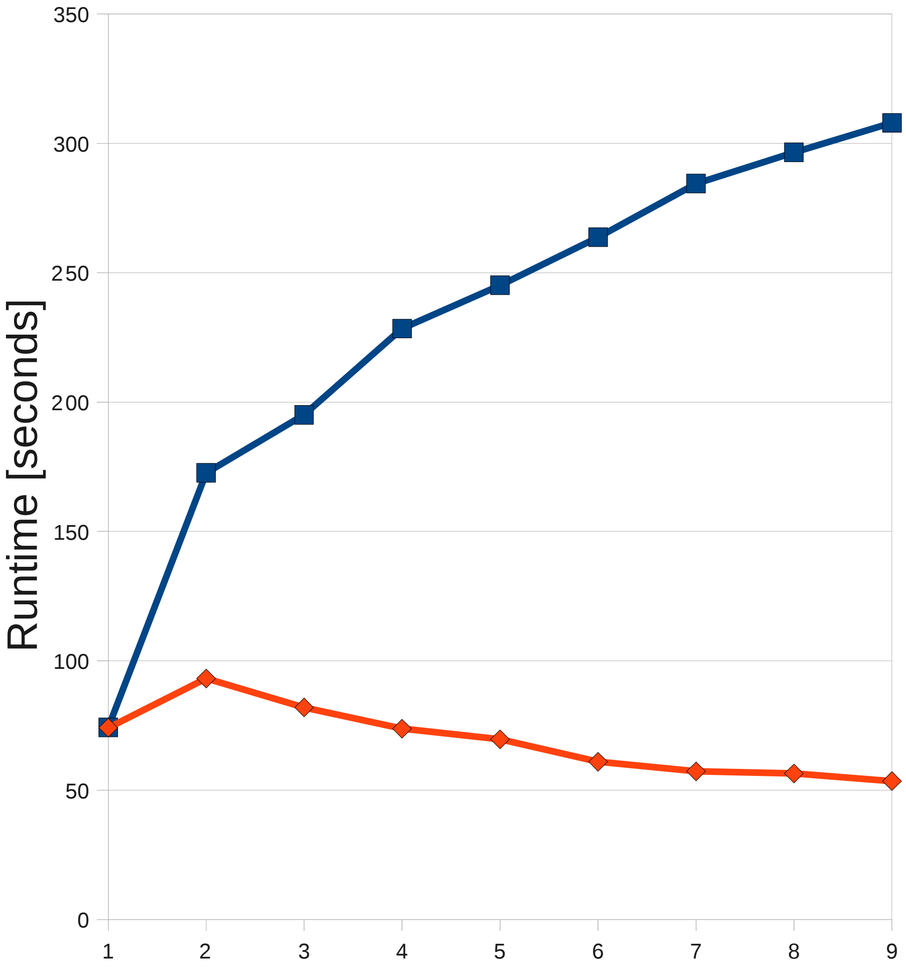
Trying To Parallelize



Somewhat Better But…



Single Core i7



Opteron NUMA

… It Is Hard To Get Right

- Many problems lurking:
  - Space overhead (many more locks when pushed down)
  - Initialization problems
    - In pthreads each mutex must be explicitly initialized
  - Definitely not possible with C++ templates
  - AB-BA locking problems
    - Need total ordering of all locks taken concurrently

C++ Specific (or: Why Not with Templates)

- Assume template classes:

```
template<mutex_t& m> portfolio;
template<mutex_t& m> bank;
```

- Even less scalable than first version because

```
   bank<some_mutex> banks[10];
```

  uses same mutex for all array elements

- Define specializations:

```
template<class Key, class T> T& map::operator(Key& x);
template<class Key, class T> T& map::operator(Key& x,
                                              mutex_t& m);
```

  Does not solve anything…

Implicit Locking Not Sufficient

- For transactions we need more complex locking

```
if (account1.mutex < account2.mutex) {
  mutex_lock(account1.mutex);
  mutex_lock(account2.mutex);
} else {
  mutex_lock(account2.mutex);
  mutex_lock(account1.mutex);
}
account1.balance -= sum;
account2.balance += sum;
if (account1.mutex < account2.mutex) {
  mutex_unlock(account2.mutex);
  …
```

Consequently

- Locking in type-generic code is either
  - Somewhat simple to use (implicit locking) and limited in application

  or

  - Hard to use (explicit, external locking) and general enough to be used in all cases
- Neither case works for automatic, implicit parallelization

We need something completely different!

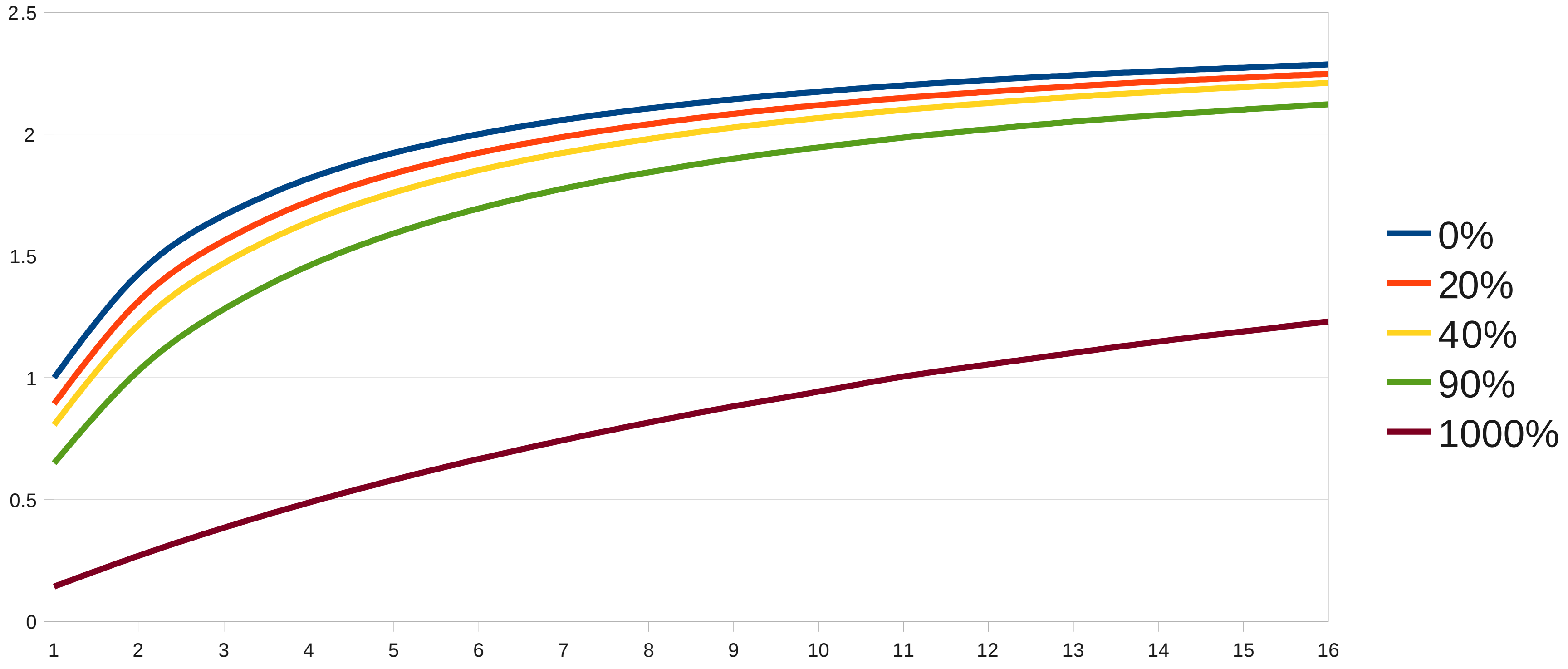
A More Realistic Formula

- Extended Amdahl's Law: overhead factors

$$S = \frac{1}{(1-P)\,(1+O_S) + \frac{P}{N}\,(1+O_P)}$$

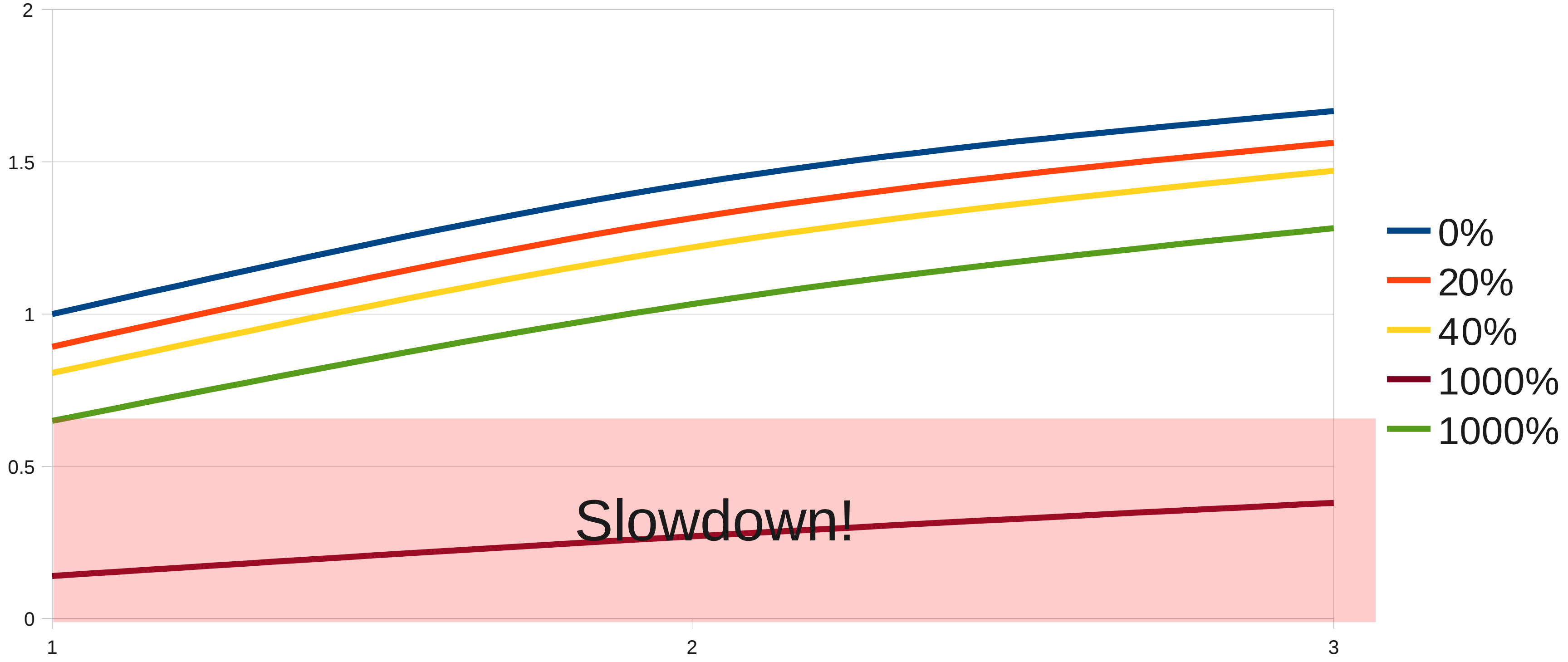
- Parallelization is not free
  - Most of the time not even for serial code
- The results are not *that* bad…

Even With Overhead (P=0.6)



- Even 40% overhead not that much slower
- Speed-up from two threads on

Even With Overhead (P=0.6)



- Even with two threads faster
- We can use technologies with overhead: STM

Implicit Locking Not Sufficient

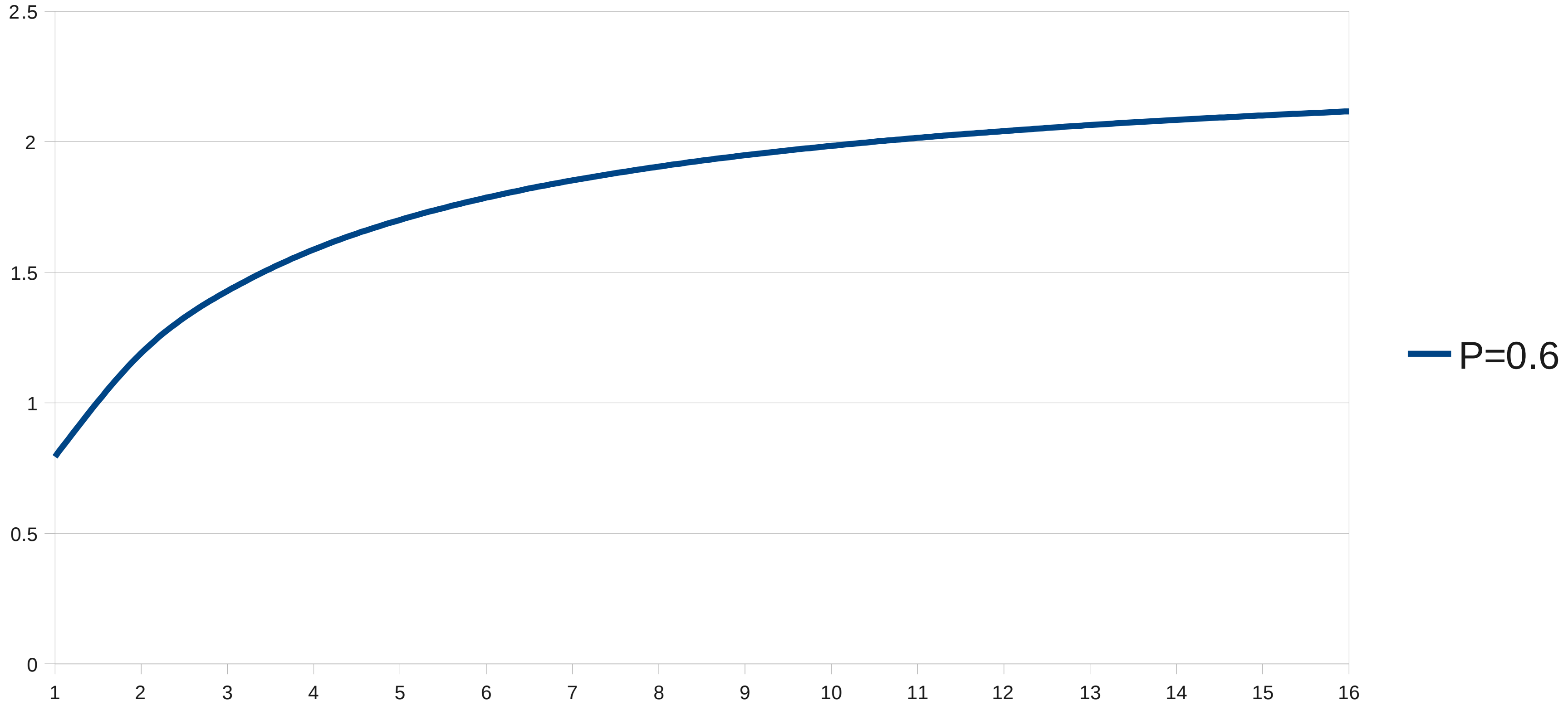
- With TM support:

```
        ~~if (account1.mutex < account2.mutex) {~~
        ~~  mutex_lock(account1.mutex);~~
        ~~  mutex_lock(account2.mutex);~~
        ~~} else {~~
        ~~  mutex_lock(account2.mutex);~~
        ~~  mutex_lock(account1.mutex);~~
        ~~}~~
__transaction {
        account1.balance -= sum;
        account2.balance += sum;
}
        ~~if (account1.mutex < account2.mutex) {~~
        ~~  mutex_unlock(account1.mutex);~~
        ~~  ...~~
```

Adjust Library

- Lots of work needed in the library
  - Make compile in TM mode without changing non-TM
  - Add `__transaction` where needed
  - Define clones when of advantage
  - Integrate with exception safety of standard library
  - Add special support for memory allocation

Performance (Projection, Sorry...)



- Assume O_S= 5% and O_P= 40%

Acknowledgement

This work has received some funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement № 216852.

Questions?

drepper@redhat.com | people.redhat.com/drepper