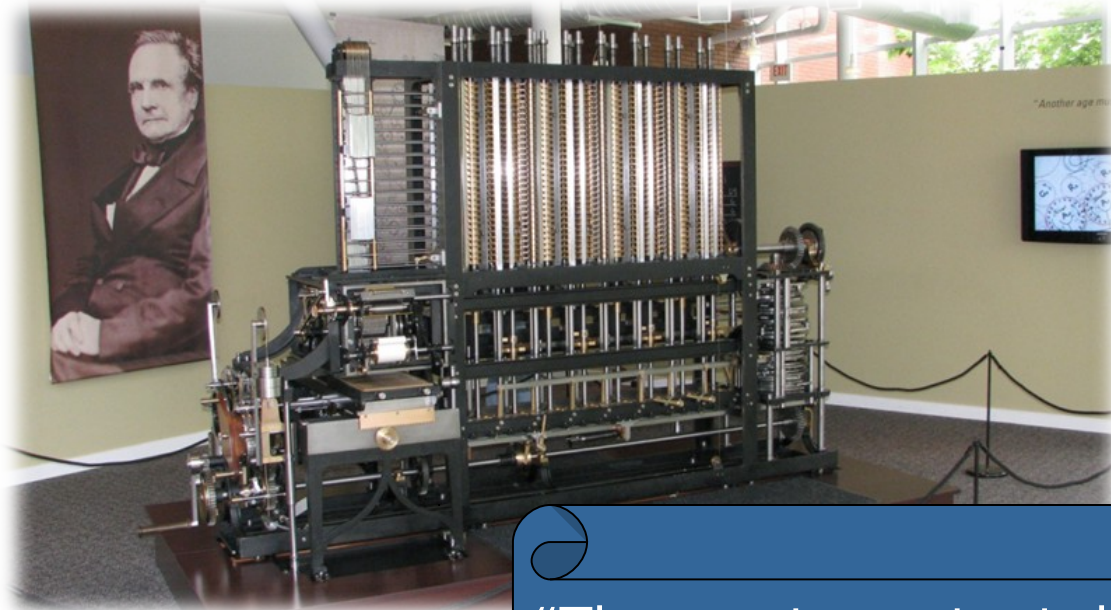




Introduction to Parallel Performance Analysis

Brian J. N. Wylie
Jülich Supercomputing Centre

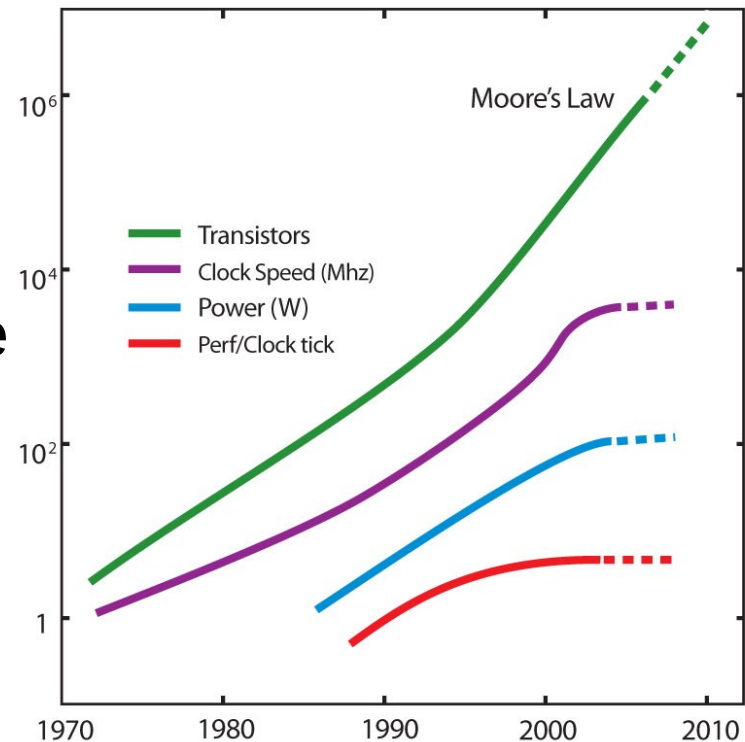
(with material taken from SC tutorials by
Bernd Mohr/JSC & Luiz de Rose/Cray)



“The most constant difficulty in contriving the engine has arisen from the desire to reduce the time in which the calculations were executed to the shortest which is possible.”

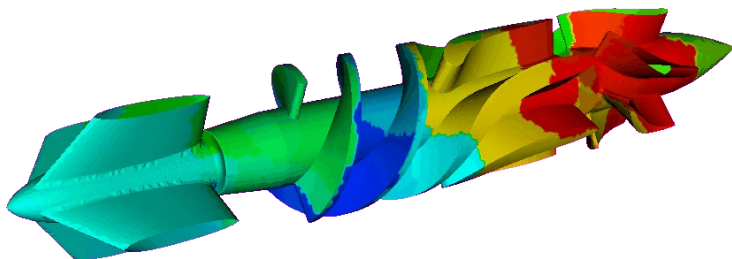
Charles Babbage
1791 – 1871

- Moore's law is still in charge, but
 - Clock rates do no longer increase
 - Performance gains only through increased parallelism
- Optimizations of applications more difficult
 - Increasing application complexity
 - Multi-physics
 - Multi-scale
 - Increasing machine complexity
 - Hierarchical networks / memory
 - More CPUs / multi-core

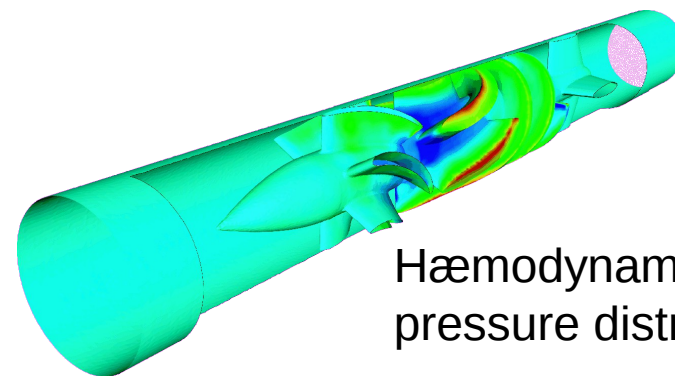


□ Every doubling of scale reveals a new bottleneck!

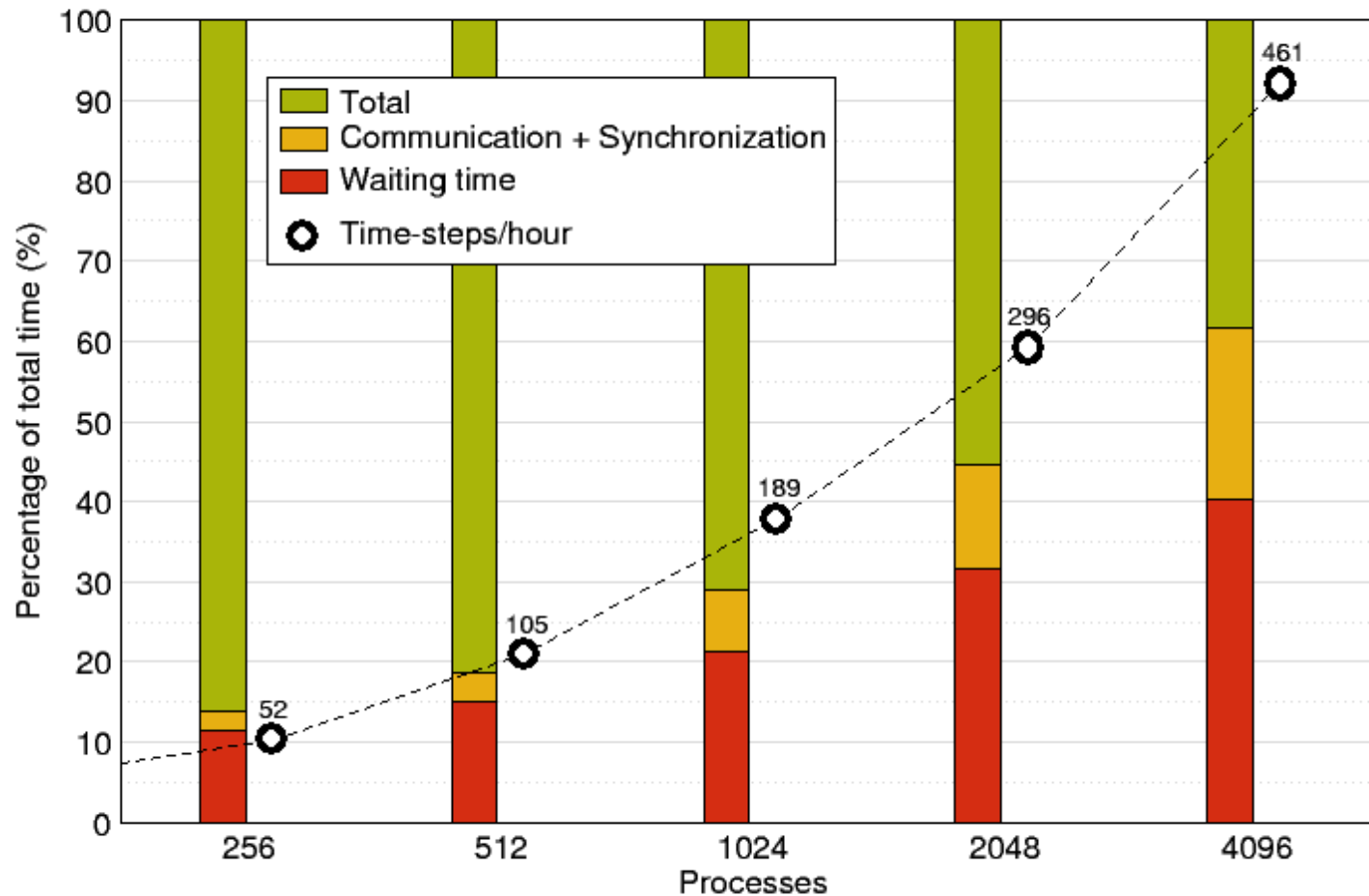
- CFD simulation of unsteady flows
 - Developed by CATS / RWTH Aachen
 - Exploits finite-element techniques, unstructured 3D meshes, iterative solution strategies
- MPI parallel version
 - >40,000 lines of Fortran & C
 - DeBaKey blood-pump data set (3,714,611 elements)



Partitioned finite-element mesh



Hæmodynamic flow
pressure distribution



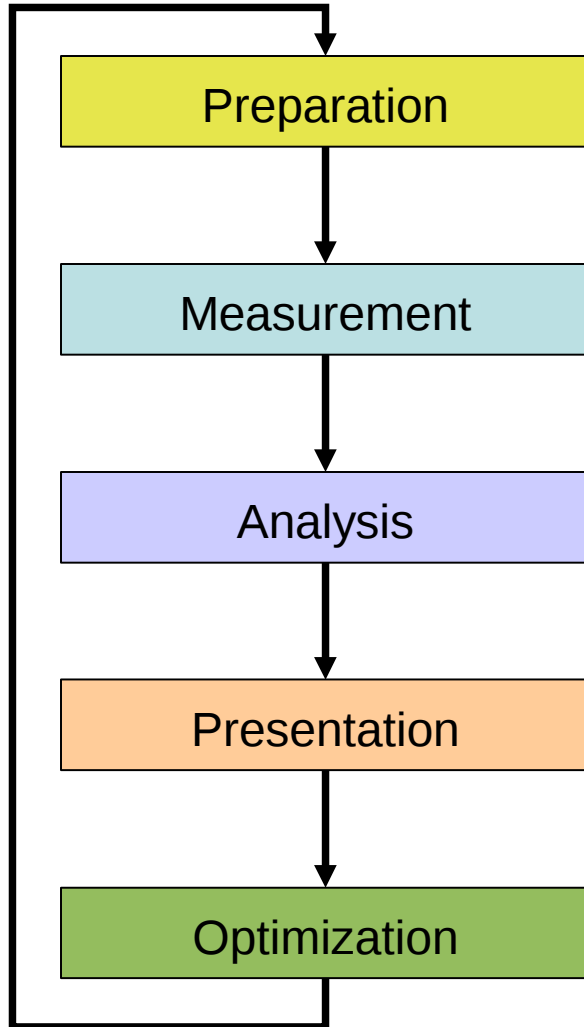
- “Sequential” factors
 - Computation
 - Choose right algorithm, use optimizing compiler
 - Cache and memory
 - Tough! Only limited tool support, hope compiler gets it right
 - Input / output
 - Often not given enough attention
- “Parallel” factors
 - Communication (i.e., message passing)
 - Threading
 - Synchronization
 - More or less understood, good tool support

- Successful tuning is a combination of
 - The right algorithms and libraries
 - Compiler flags and directives
 - Thinking !!!
- Measurement is better than guessing
 - To determine performance bottlenecks
 - To validate tuning decisions and optimizations
 - After each step!

“We should forget about small efficiencies,
say 97% of the time: premature optimization
is the root of all evil.”

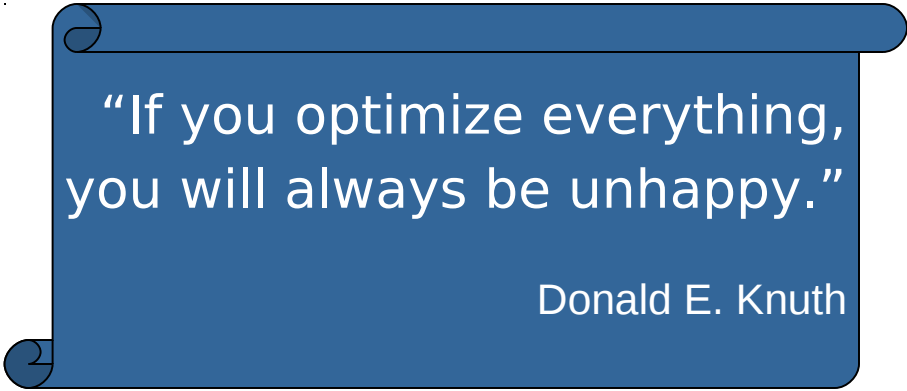
C. A. R. Hoare

- It's easier to optimize a slow correct program than to debug a fast incorrect one
 - *Nobody cares how fast you can compute the wrong answer...*



- Prepare application, insert extra code (probes/hooks)
- Collection of data relevant to performance analysis
- Calculation of metrics, identification of performance metrics
- Visualization of results in an intuitive/understandable form
- Elimination of performance problems

- Programs typically spend 80% of their time in 20% of the code
- Programmers typically spend 20% of their effort to get 80% of the total speedup possible for the application
 - ▮ *Know when to stop!*
- Don't optimize what does not matter
 - ▮ *Make the common case fast!*

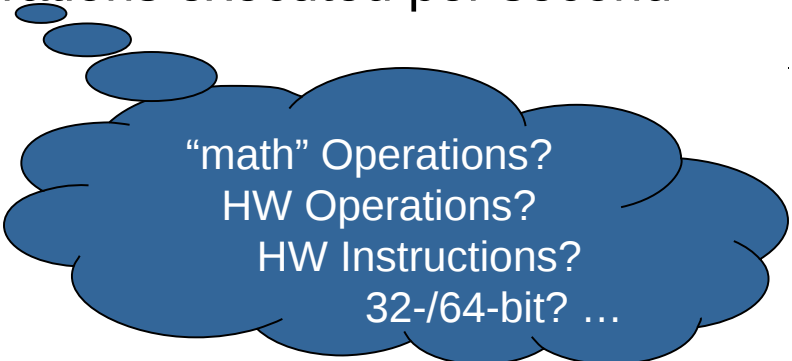


“If you optimize everything,
you will always be unhappy.”

Donald E. Knuth

- What can be measured?
 - A **count** of how many times an event occurs
 - E.g., the number of MPI point-to-point messages sent
 - The **duration** of some time interval
 - E.g., the time spent these send calls
 - The **size** of some parameter
 - E.g., the number of bytes transmitted by these calls
- Derived metrics
 - E.g., rates / throughput
 - Needed for normalization

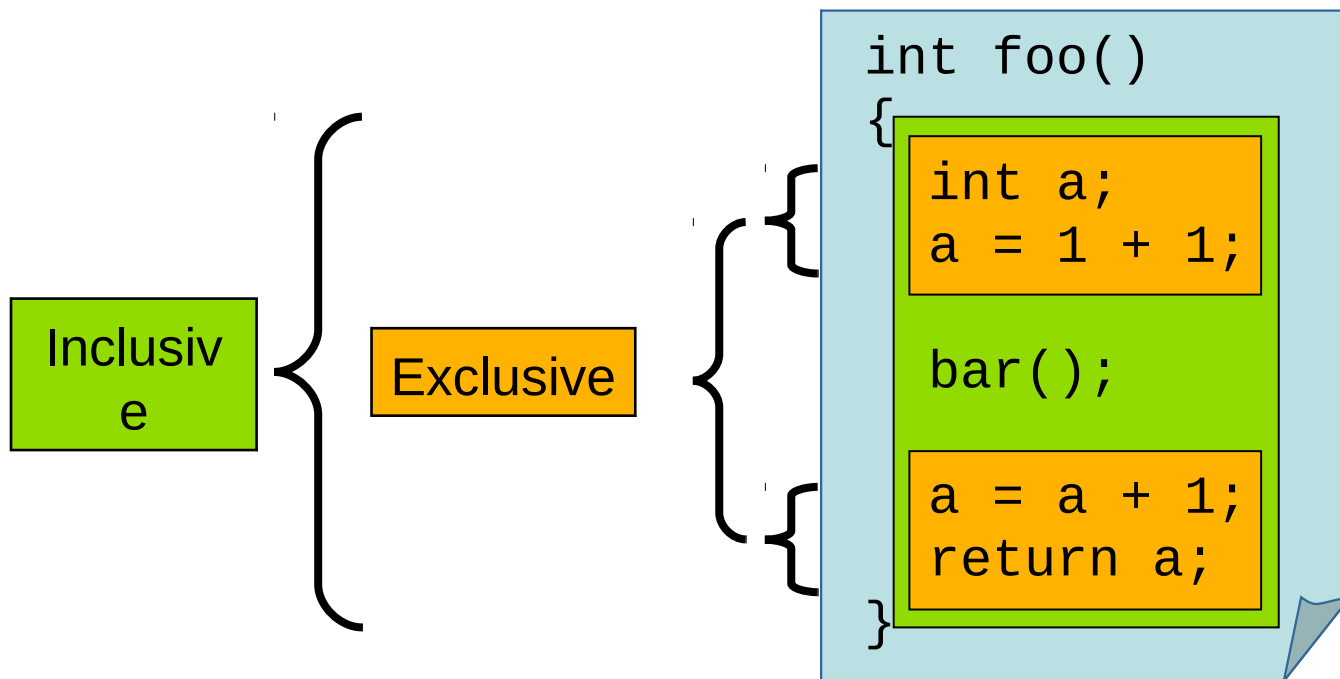
- Execution time
- Number of function calls
- CPI
 - Clock ticks per instruction
- MFLOPS
 - Millions of floating-point operations executed per second



“math” Operations?
HW Operations?
HW Instructions?
32-/64-bit? ...

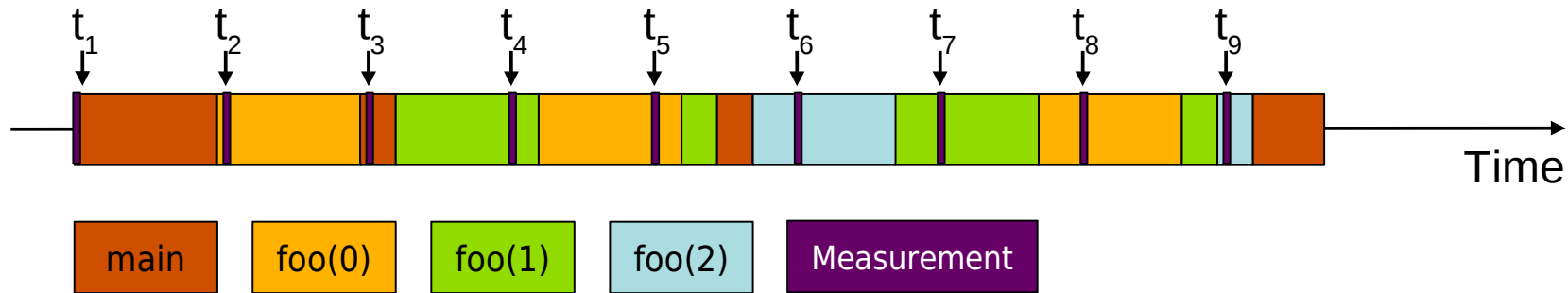
- Wall-clock time
 - Includes waiting time: I/O, memory, other system activities
 - In time-sharing environments also the time consumed by other applications
- CPU time
 - Time spent by the CPU to execute the application
 - Does not include time the program was context-switched out
 - Problem: Does not include inherent waiting time (e.g., I/O)
 - Problem: Portability? What is user, what is system time?
- Problem: Execution time is non-deterministic
 - Use mean or minimum of several runs

- Inclusive
 - Information of all sub-elements aggregated into single value
- Exclusive
 - Information cannot be split up further



- When are performance measurements triggered?
 - Sampling
 - Code instrumentation

- How is performance data recorded?
 - Profiling / Runtime summarization
 - Tracing



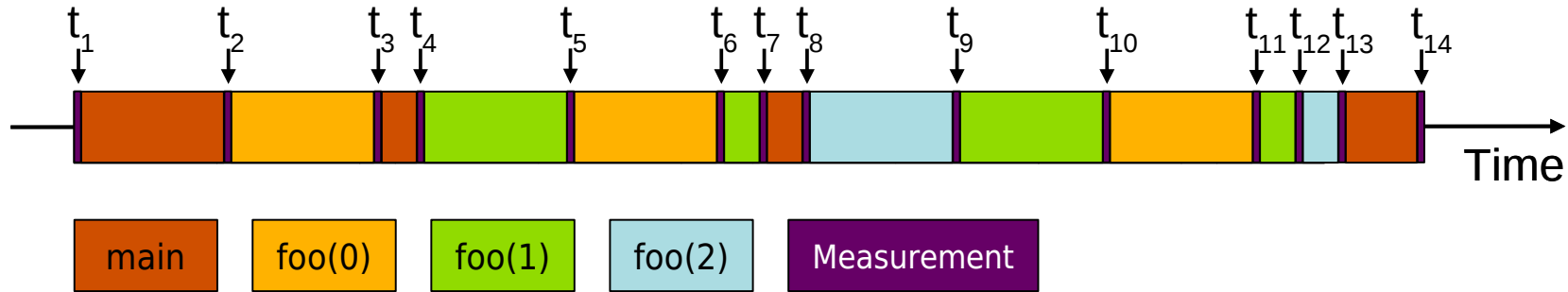
```
int main()
{
    int i;

    for (i=0; i < 3; i++)
        foo(i);

    return 0;
}

void foo(int i)
{
    if (i > 0)
        foo(i - 1);
}
```

- Running program is interrupted periodically
 - Timer interrupt, OS signal, or HWC overflow
 - Service routine examines return-address stack
 - Addresses are mapped to routines using symbol table information
- **Statistical** inference of program behaviour
 - Not very detailed information on highly volatile metrics
 - Requires long-running applications
- Works with unmodified executable, but symbol table information is generally recommended



```
int main()
{
    int i;
    Enter("main");
    for (i=0; i < 3; i++)
        foo(i);
    Leave("main");
    return 0;
}

void foo(int i)
{
    Enter("foo");
    if (i > 0)
        foo(i - 1);
    Leave("foo");
}
```

- Measurement code is inserted such that every interesting event is **directly** captured
 - Can be done in various ways
- Advantage:
 - Much more detailed information
- Disadvantage:
 - Processing of source-code / executable necessary
 - Large relative overheads for small functions

- **Static** instrumentation
 - Program is instrumented prior to execution
- **Dynamic** instrumentation
 - Program is instrumented at runtime
- Code is inserted
 - Manually
 - Automatically
 - By a preprocessor / source-to-source translation tool
 - By a compiler
 - By linking against a pre-instrumented library / runtime system
 - By binary-rewrite / dynamic instrumentation tool

- Accuracy
 - Perturbation
 - Measurement alters program behavior
 - E.g., memory access pattern
 - Intrusion overhead
 - Measurement itself needs time and thus lowers performance
 - Accuracy of timers & counters
- Granularity
 - How many measurements?
 - How much information / work during each measurement?

□ *Tradeoff: Accuracy vs. Expressiveness of data*

- When are performance measurements triggered?
 - Sampling
 - Code instrumentation

- How is performance data recorded?
 - Profiling / Runtime summarization
 - Tracing

- Recording of aggregated information
 - Time
 - Counts
 - Function calls
 - Hardware counters
- Over program and system entities
 - Functions, call sites, basic blocks, loops, ...
 - Processes, threads

□ *Profile = summation of events over time*

- Flat profile
 - Shows distribution of metrics per function / instrumented region
 - Calling context is not taken into account
- Call-path profile
 - Shows distribution of metrics per call path
 - Sometimes only distinguished by partial calling context (e.g., two levels)
- Special-purpose profiles
 - Focus on specific aspects, e.g., MPI calls or OpenMP constructs

- Recording information about significant points (events) during execution of the program
 - Enter / leave of a region (function, loop, ...)
 - Send / receive a message, ...
 - Save information in event record
 - Timestamp, location, event type
 - Plus event-specific information (e.g., communicator, sender / receiver, ...)
 - Abstract execution model on level of defined events
- *Event trace = Chronologically ordered sequence of event records*

Event tracing

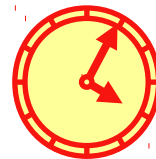
Process A

```
void foo() {  
    ...  
    send(B, tag, buf);  
    ...  
}
```

MONITOR



synchronize(d)



Process B

```
void bar() {  
    ...  
    recv(A, tag, buf);  
    ...  
}
```

MONITOR

Local trace A

...		
58	ENTER	1
62	SEND	B
64	EXIT	1
...		

1	foo
...	

Local trace B

...		
60	ENTER	1
68	RCV	A
69	EXIT	1
...		

1	bar
...	

Global trace

...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RCV	A
69	B	EXIT	2
...			

merge

unify

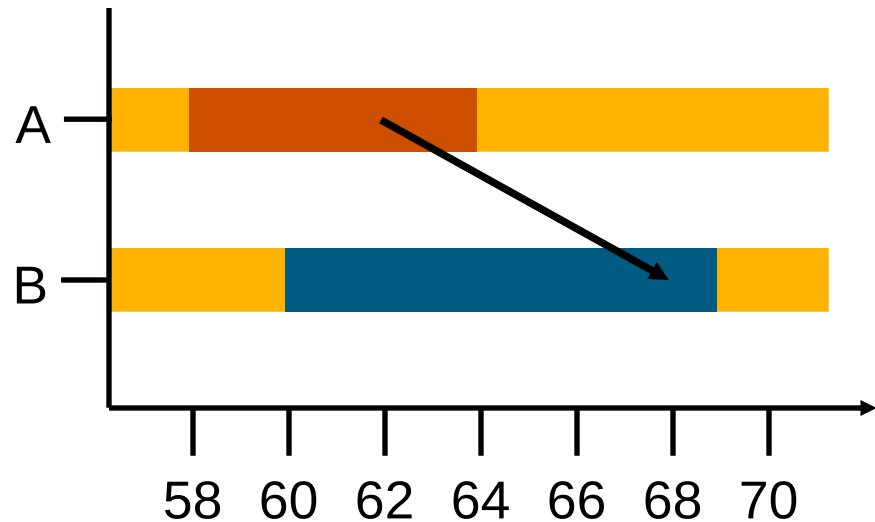
1	foo
2	bar
...	

Example: Time-line visualization

1	foo
2	bar
3	...



...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			



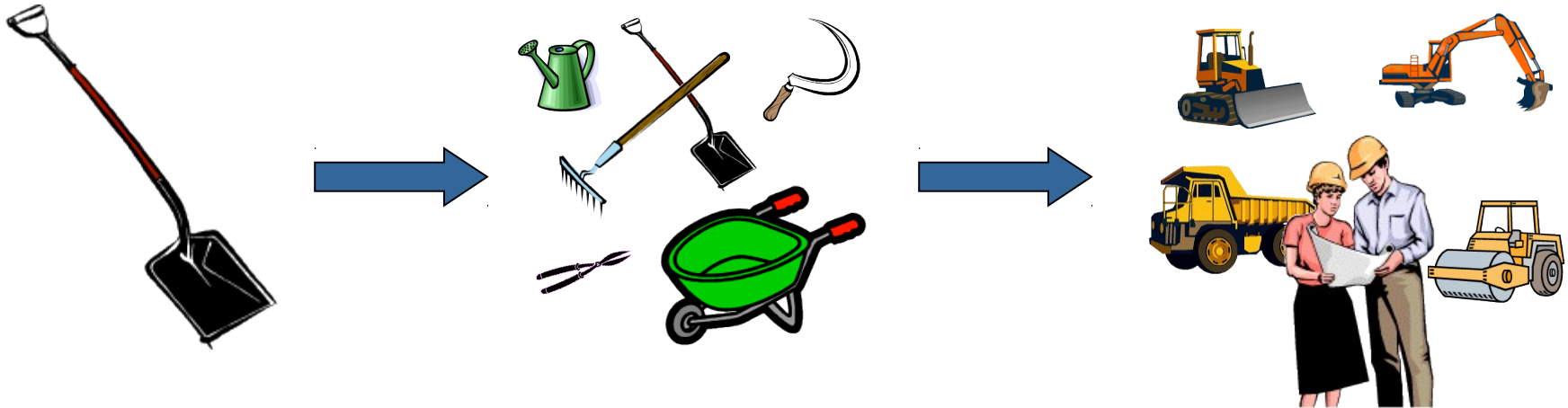
■ Tracing advantages

- Event traces preserve the **temporal** and **spatial** relationships among individual events (□ context)
- Allows reconstruction of **dynamic** application behavior on any required level of abstraction
- Most general measurement technique
 - Profile data can be reconstructed from event traces

■ Disadvantages

- Traces can become very large
- Writing events to file at runtime causes perturbation
- Writing tracing software is complicated
 - Event buffering, clock synchronization, ...

- Do I have a performance problem at all?
 - Time / speedup / scalability measurements
- **What** is the main bottleneck (computation / communication)?
 - MPI / OpenMP / flat profiling
- **Where** is the main bottleneck?
 - Call-path profiling, detailed basic block profiling
- **Why** is it there?
 - Hardware counter analysis, trace selected parts to keep trace size manageable
- Does my code have scalability problems?
 - Load imbalance analysis, compare profiles at various sizes function-by-function



□ *A combination of different methods, tools and techniques is typically needed!*

- Measurement
 - Sampling / instrumentation, profiling / tracing, ...
- Instrumentation
 - Source code / binary, manual / automatic, ...
- Analysis
 - Statistics, visualization, automatic analysis, data mining, ...