从程序优化角度，我们的课程安排

- Week 1: Tuning for In-core programming
- Week 2: Tools: Intel compiler and Vtune
- Week 3: Tuning for Parallel programming
- Week 4: Show case

并行程序性能分析与优化

本主题的内容

- 并行程序性能优化
  - 如何来分析并行程序的性能？
  - 并行程序有哪些性能问题？
  - 如何优化并行程序性能？
  - 哪些工具帮助我们诊断并行程序的性能问题？

分析的问题

- 如何量度一个程序的性能？
- 如何衡量一个并行程序的好坏？

并行程序性能指标和性能模型

通用的性能评价指标

- 程序执行时间
  是指用户的响应时间 (访问磁盘和访问存储器的时间， CPU时间， I/O时间以及操作系统的开销)
  Linux: gettimeofday(us) / clock_gettime (ns)
  - Windows: QueryPerformanceFrequency / QueryPerformanceCounter
  - MPI_Wtime/MPI_Wtick
- 获取并行程序时间
  1. Barrier
  2. Start Timer
  3. Run Program
  4. End Timer
  5. Max(EndTime[i]-StartTime[i])

Speedup

- The *speedup* of a parallel application is

$$\text{Speedup}(p) = \text{Time}(1)/\text{Time}(p)$$

- Where
  - Time(1) = execution time for a single processor and
  - Time(p) = execution time using p parallel processors
- If Speedup(p) = p we have *perfect speedup* (also called *linear scaling)*

- As defined, speedup compares an application with itself on one and on p processors, but it is more useful to compare
  - The execution time of the best serial application on 1 processor

versus

  - The execution time of best parallel algorithm on p processors

Superlinear Speedup

Question: can we find "*superlinear*" speedup, that is

Speedup(p) > p ?

- Choosing a bad "baseline" for T(1)
  - Old serial code has not been updated with optimizations
  - Avoid this, and always specify what your baseline is
- Shrinking the problem size per processor
  - May allow it to fit in small fast memory (cache)
- Application is not deterministic
  - Amount of work varies depending on execution order
  - Search algorithms have this characteristic

Efficiency

- The *parallel efficiency* of an application is defined as

$$\text{Efficiency}(p) = \text{Speedup}(p)/p$$

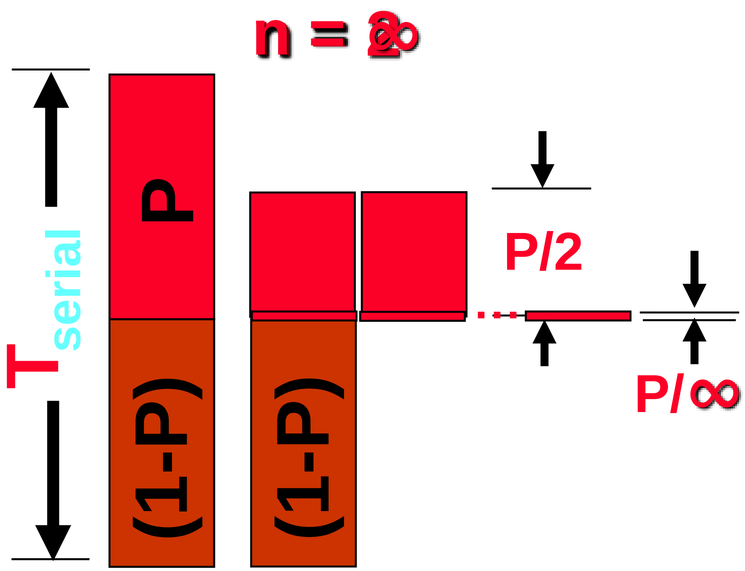
    - Efficiency(p) <= 1
    - For perfect speedup Efficiency (p) = 1
- We will rarely have perfect speedup.
    - Lack of perfect parallelism in the application or algorithm
    - Imperfect load balancing (some processors have more work)
    - Cost of communication
    - Cost of contention for resources, e.g., memory bus, I/O
    - Synchronization time
- Understanding why an application is not scaling linearly will help finding ways improving the applications performance on parallel computers.

并行程序性能分析 – 性能模型

- 加速比定律
  - 并行系统的加速比是指对于一个给定的应用，并行算法（或并行程序）的执行速度相对于串行算法（或串行程序）的执行速度加快了多少倍。
  - **Amdahl 定律**
  - Gustafson定律
- 可扩放性评测标准
  - 等效率度量标准

Amdahl's Law

- Describes the upper bound of parallel execution speedup



0.5 + 0.25

$$T_{parallel} = \{(1-P) + P/n\}\, T_{serial}$$

n = number of processors

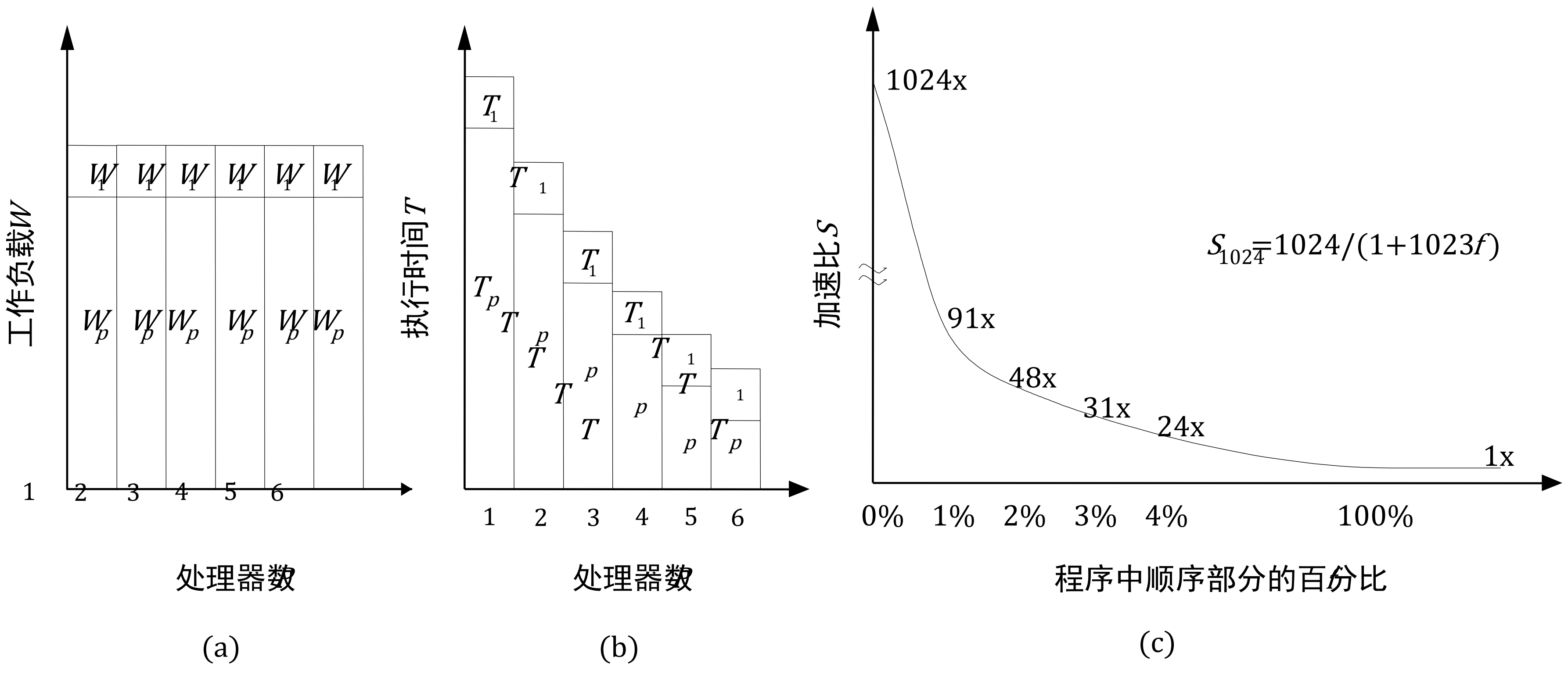
$$Speedup = T_{serial} / T_{parallel}$$ 1.0/0.55 = 2.033

Serial code limits speedup

Amdahl 定律

- P：处理器数；
- W：问题规模(计算负载、工作负载，给定问题的总计算量)；
  - W_s：应用程序中的串行分量，f是串行分量比例（$f = W_s/W$，$W_s=W_1$）；
  - W_P：应用程序中可并行化部分，1-f为并行分量比例；
  - $W_s + W_p = W$；
- $T_s=T_1$：串行执行时间，T_p：并行执行时间；
- S：加速比，E：效率；
- 出发点：
  - 固定不变的计算负载；
  - 固定的计算负载分布在多个处理器上并可以有效并行
  - 增加处理器加快执行速度，从而达到了加速的目的。

Amdahl定律



工作负载W
处理器数
(a)
执行时间T
处理器数
(b)
加速比S
1024x
S_{1024}=1024/(1+1023f)
91x
48x
31x
24x
1x
0%
1%
2%
3%
4%
100%
程序中顺序部分的百分比
(c)

Gustafson定律

- 出发点：
  - 对于很多大型计算，精度要求很高，即在此类应用中精度是个关键因素，而计算时间是固定不变的。此时为了提高精度，必须加大计算量，相应地亦必须增多处理器数才能维持时间不变
  - 除非学术研究，在实际应用中没有必要固定工作负载而计算程序运行在不同数目的处理器上，增多处理器必须相应地增大问题规模才有实际意义
- Gustafson加速定律：

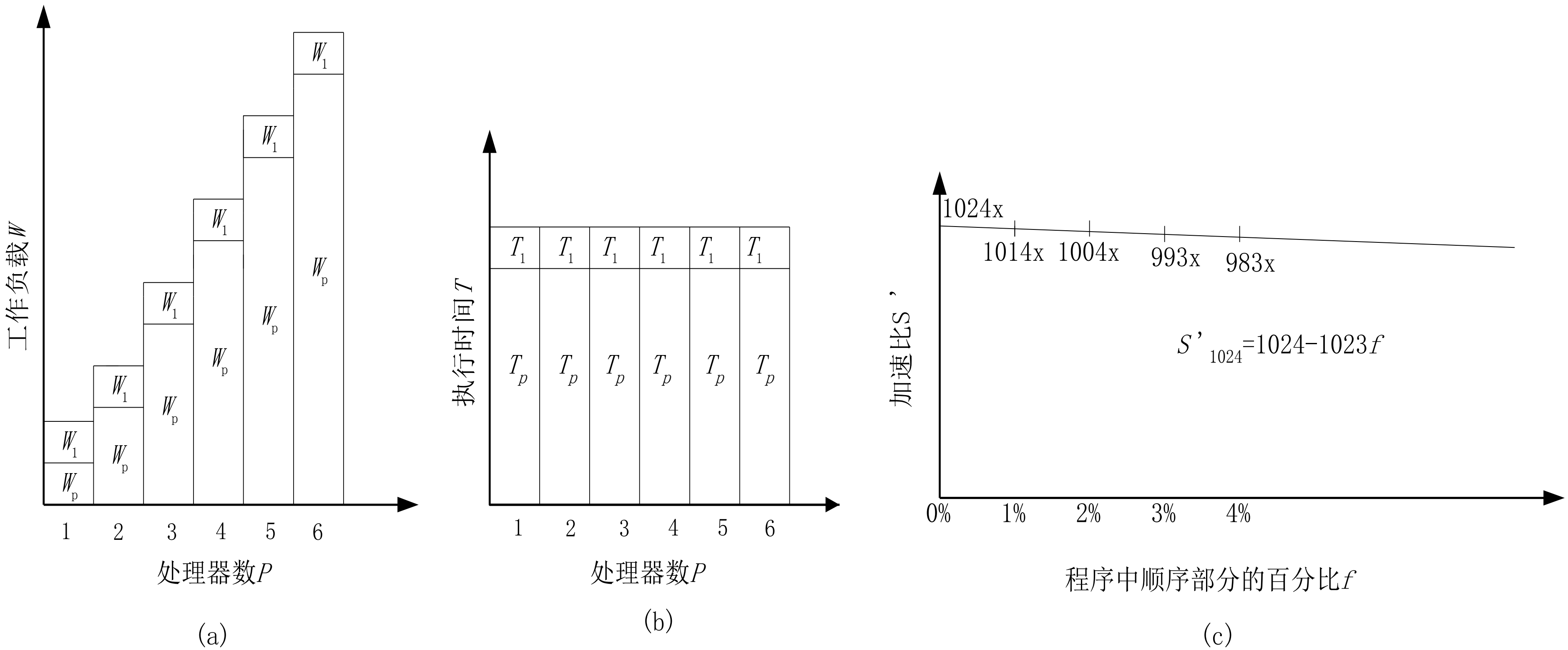
$$S' = \frac{W_S + pW_P}{W_S + p \cdot W_P / p} = \frac{W_S + pW_P}{W_S + W_P}$$

$$S' = f + p(1-f) = p + f(1-p) = p - f(p-1)$$

- 并行开销W_o：

$$S' = \frac{W_S + pW_P}{W_S + W_P + W_O} = \frac{f + p(1-f)}{1 + W_O / W}$$

Gustafson定律



增加问题的规模有利于提高加速的因素：

较大的问题规模可提供较高的并发度；

额外开销的增加可能慢于有效计算的增加；

算法中的串行分量比例不是固定不变的（串行部分所占的比例随着问题规模的增大而缩小）。

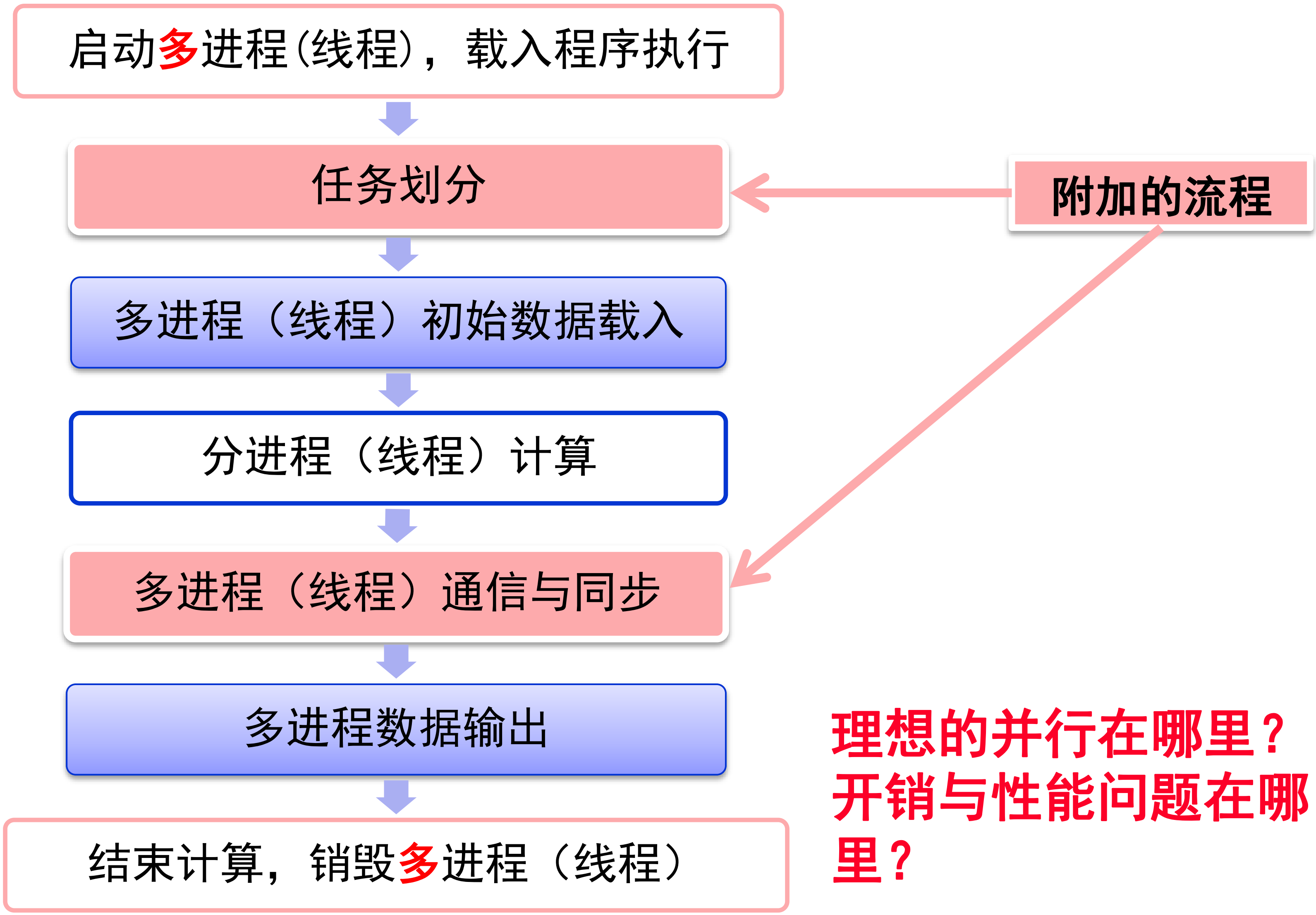
加速比分析

- 影响加速比的因素：处理器数与问题规模
  - 加大的处理器数超过了算法中的并发程度
  - 求解问题中的串行分量
  - 并行处理所引起的额外开销（通信、等待、竞争、冗余操作和同步等）

粗略的串行程序流程

启动进程(线程)，载入程序执行	启动单进程(线程)
读入初始数据	串行IO
实施计算	串行计算
输出数据	串行IO
结束计算，销毁进程（线程）	销毁单进程(线程)

粗略的并行程序流程



理想的并行在哪里？开销与性能问题在哪里？

常见的并行开销有哪些？

1. 创建和销毁并行进程、**线程**的开销

- 创建和销毁进程本身是高开销的工作
  - PowerPC 700MHz(每个周期 15ns 执行 4flops; 创建一个进程1.4ms，可执行 372,000flops)
- 创建和销毁多个进程的开销在系统中随进程数增加
  - 启动4096进程可能需要s级时间

常见的并行开销有哪些？

- 通信开销是并行开销的主要部分
  - **多机间的通信开销相对于计算很大**
  - Cray T3E
  - FP peak: 900 Mflops ➔ 1.11 nanoseconds/flop
  - Communication using MPI (Message Passing Interface):
    - Latency: 3 microseconds (~ 2702 FPs)
    - Bandwidth: 120 MB/s (~ 60 FPs/double-word)
  - IBM SP Power3
  - FP peak: 1.5 Gflops ➔ 0.67 nanoseconds/flop
  - Communication using MPI:
    - Latency: 8 microseconds (~ 11940 FPs)
    - Bandwidth: 347 MB/s (~ 35 FPs/double-word)
  - Linux Pentium3 cluster (J. Riedy, UC Berkeley)
  - FP peak: 700 Mflops ➔ 1.43 nanoseconds/flop
  - Communication
    - Latency: 80 microseconds (~ 55944 FPs)
    - Bandwidth: 40 MB/s (~ 140 FPs/double-word)

量度点对点通信开销

- *Transfer time (n)* $= T_0 + n/B$
  - useful for message passing, memory access
- As *n* increases, bandwidth approaches asymptotic rate *B*
- How quickly it approaches depends on T_0 (latency)

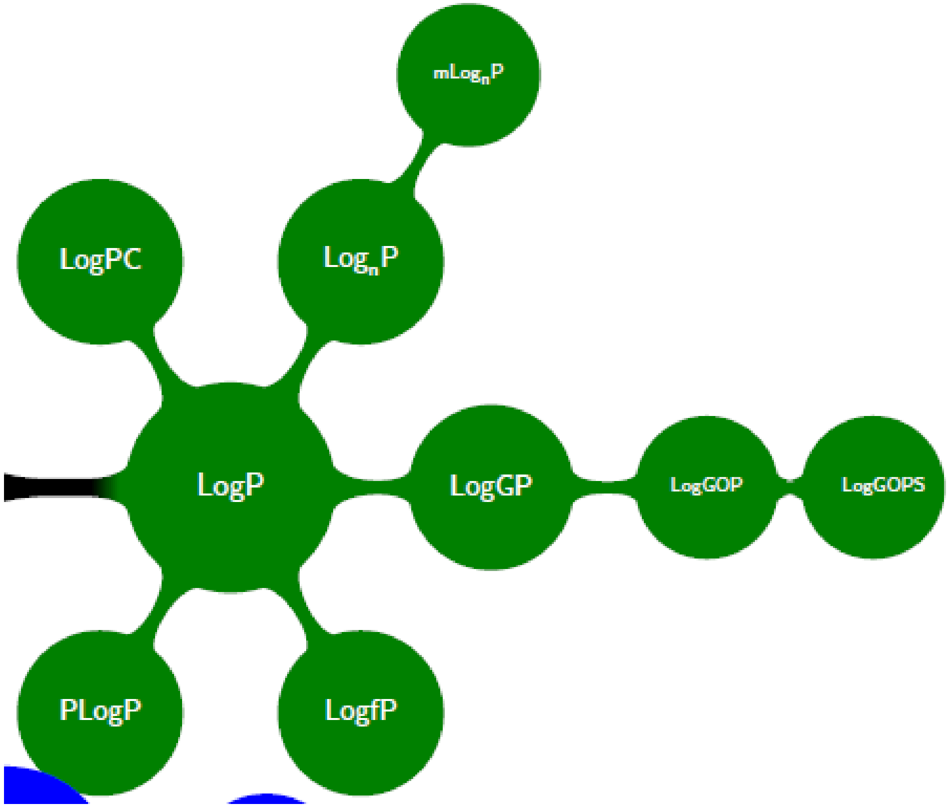
量度群集通信开销

- 典型的群集通信
  - 广播（Broadcasting）：处理器0发送m个字节给所有的n个处理器；
  - 收集（Gather）：处理0接收所有n个处理器发来在消息，所以处理器0最终接收了m n个字节；
  - 散射（Scatter）：处理器0发送了m个字节的不同消息给所有n个处理器，因此处理器0最终发送了m n个字节；
  - 全交换（Total Exchange）：每个处理器均彼此相互发送m个字节的不同消息给对方，所以总通信量为mn^2个字节；
  - $T(m, n) = t_0(n) + m / r_\infty(n)$

通信性能依平台和实现不同而变化，可以参考http://mvapich.cse.ohio-state.edu/benchmarks/或者IMB完成MPI通信测试

It is not so easy… …

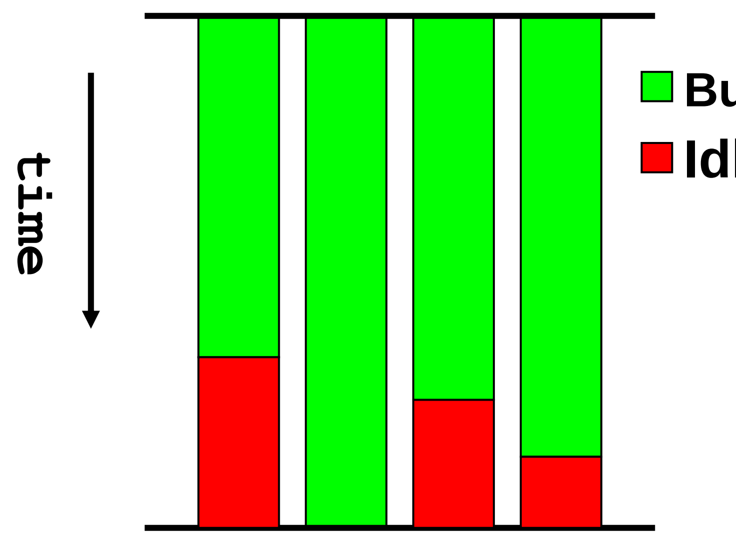
- 通信模型参数会因很多因素不同而变化
  - 同时通信的进程数
  - 同时发送的消息数
  - 消息的大小
  - 网络的拓扑结构
  - 网络的拥挤程度
  - 不同的MPI实现
  - 群集消息通信算法
  - 之前发送的消息情况
  - … …



Threading Issues for performance

- Data Races introduce performance issues
  - Concurrent access of same variable by multiple threads
    - Synchronization
      - Share data access must be coordinated
      - Explicit and implicit Barrier in OMP
      - Be aware of memory model and implicit barrier (parallel/for/single)

Fortran	C/C++
BARRIER END PARALLEL CRITICAL and END CRITICAL END DO END SECTIONS END SINGLE ORDERED and END ORDERED	barrier parallel - upon entry and exit critical - upon entry and exit ordered - upon entry and exit for - upon exit sections - upon exit single - upon exit

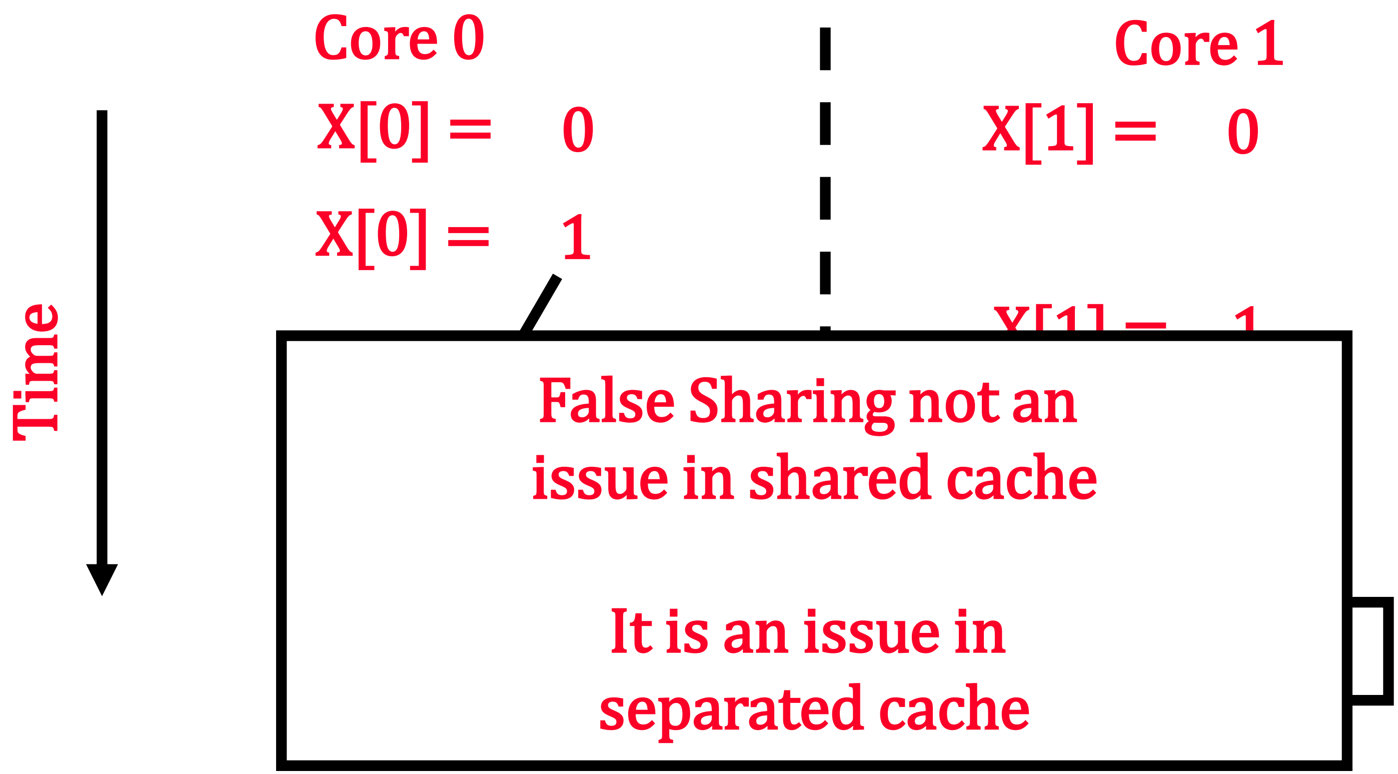


Threading Issues for performance

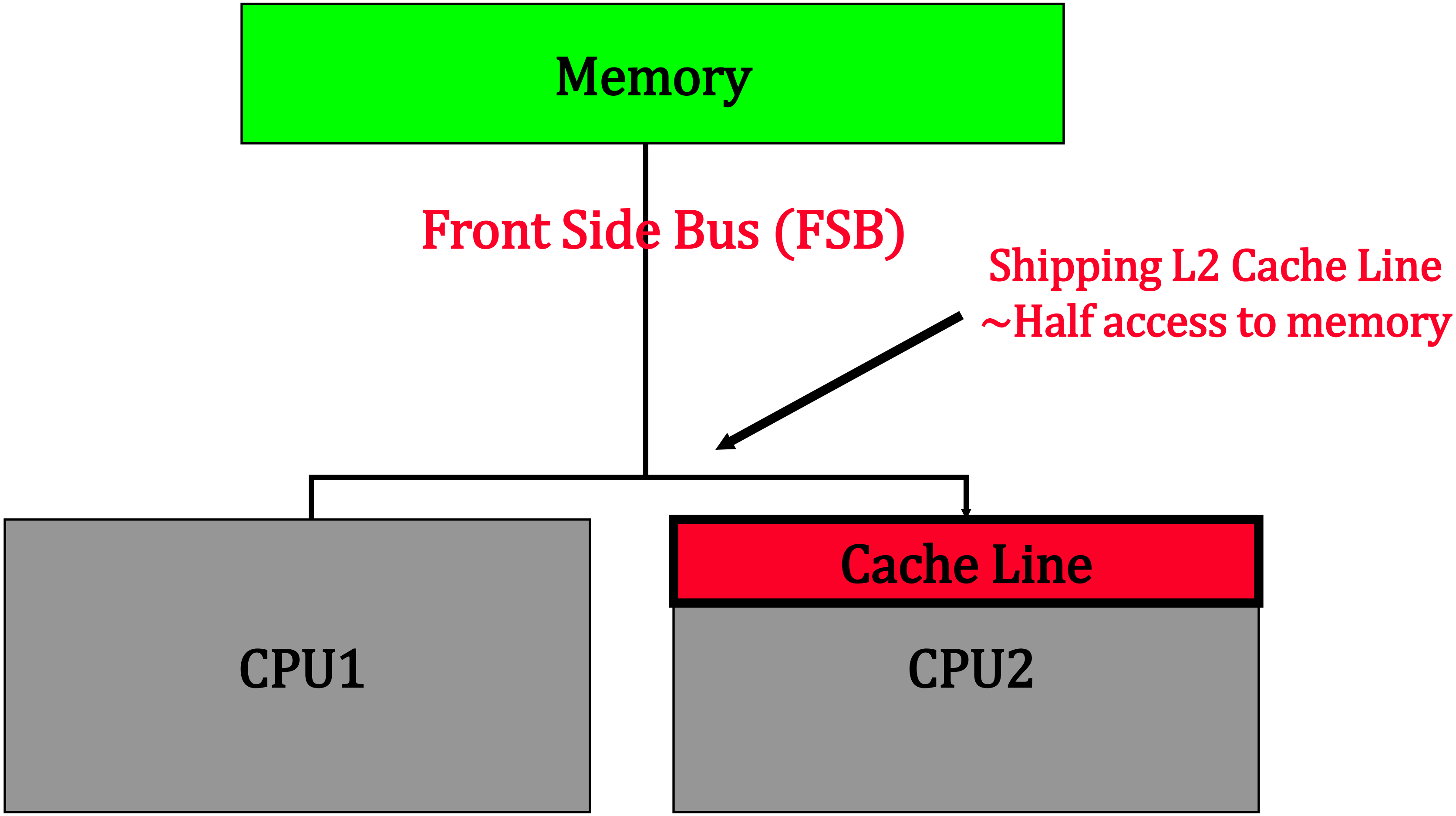
- Data Races introduce performance issues
  - Mutex
    - Atomic /critical /reduction will lead to serial computation
    - 优化策略：减小串行范围，减少同步次数
  - Dead Locks
    - Indefinite wait for resources, caused by locking hierarchy in threads
  - False Sharing
    - Threads writing different data on the same cache line

False Sharing

- Performance issue in programs where cores may write to different memory addresses BUT in the same cache lines
- Known as Ping-Ponging – Cache line is shipped between cores

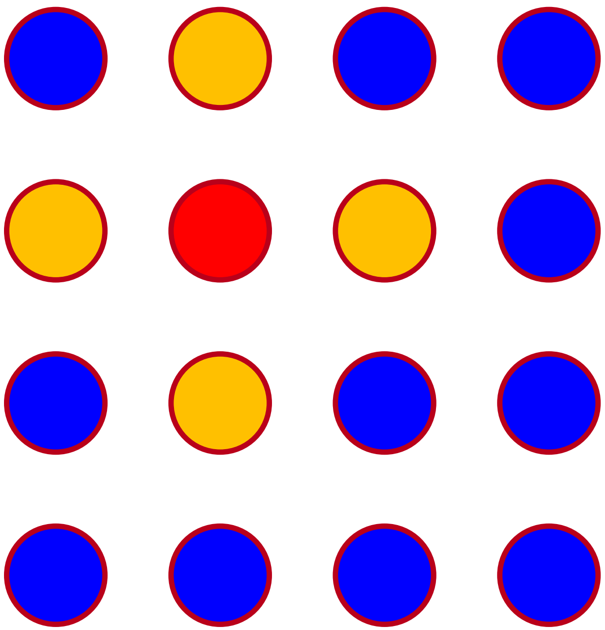


With a separated cache



常见的并行开销还有哪些？

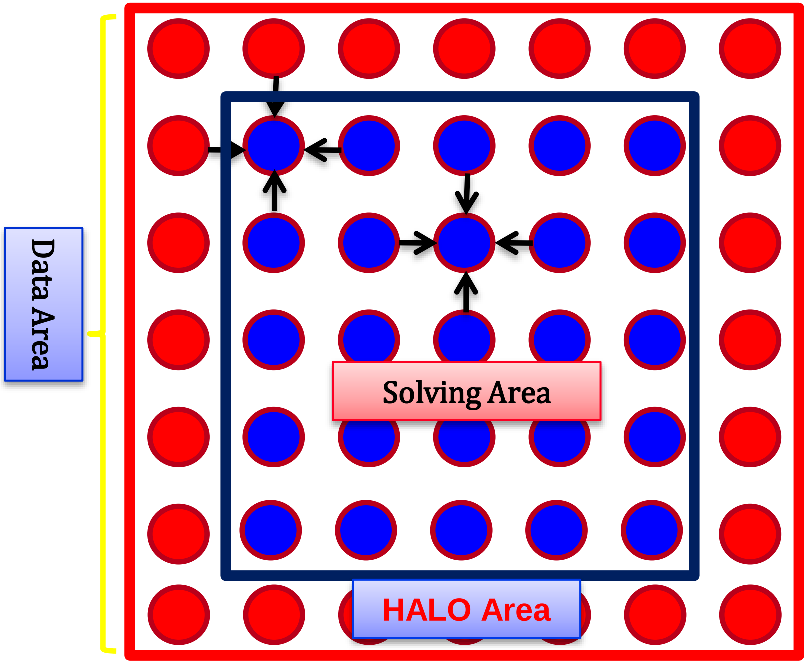
- 并行化过程中引入的空间和相应的时间开销



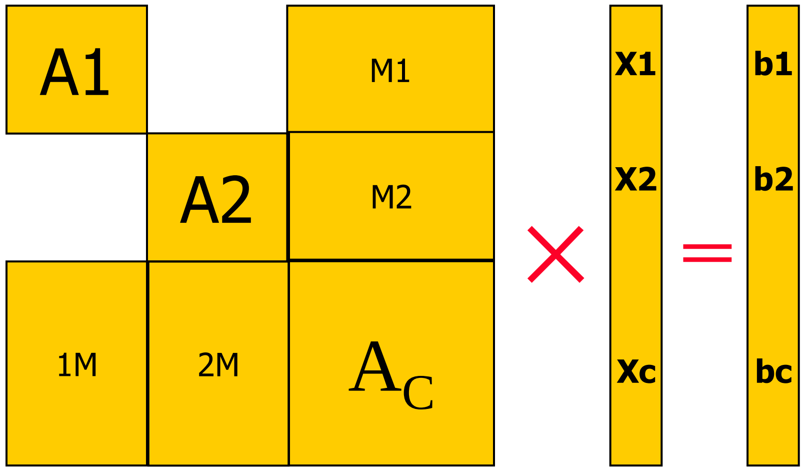
```
DO J=1,N
      DO I=1,N
            B(I,J)=0.25*(A(I-1,J)+A(I+1,J)+A(I,J+1)+A(I,J-1))
      END DO
END DO
DO J=1,N
      DO I=1,N
            A(I,J)=B(I,J)
      END DO
END DO
```

常见的并行开销还有哪些？

- 并行化过程中引入的空间和相应的时间开销
    - 消息缓冲区准备
    - 交叠数据的分配和使用



例子



Step 1:

Proc 1: Compute T1=-1M * inv(A1) * M1; Tb1= -1M * inv(A1) * b1 $T_{p1\text{-}s1}$

Proc 2: Compute T2=-2M * inv(A2) * M2; Tb2= -2M * inv(A2) * b2 $T_{p2\text{-}s1}$

Step 2:

Communicate T1/Tb1, T2/Tb2 to Proc C $T_{p1\text{-}C}$ / $T_{p2\text{-}C}$

Step 3:

Proc C: Compute bc by (AC-T1-T2)*Xc=bc-Tb1-Tb2 $T_{C\text{-}s3}$

Step 4:

Communicate Xc to Proc 1, Proc2 $T_{C\text{-}p1}$ / $T_{C\text{-}p2}$

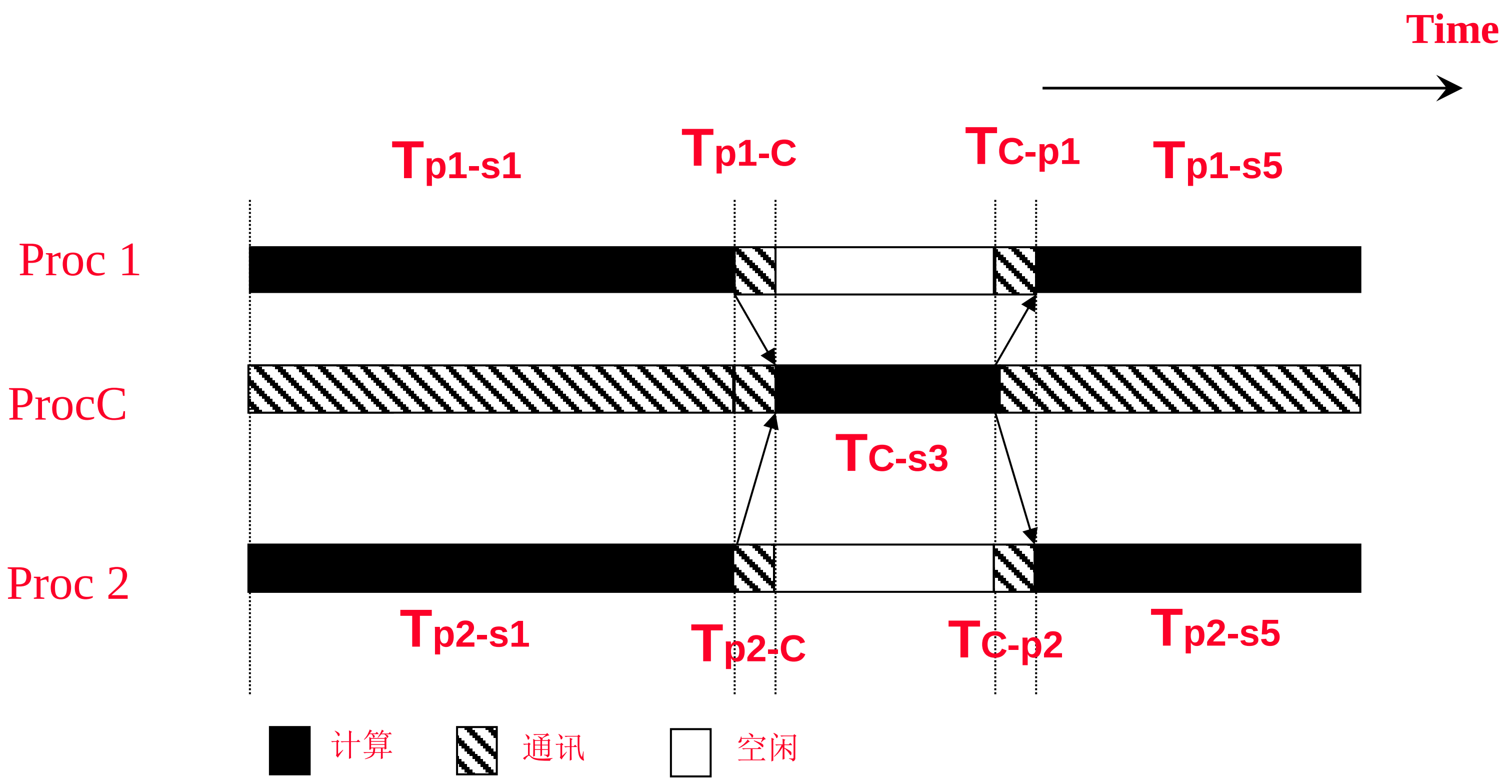
Step 5:

Proc 1: Compute X1 $T_{p1\text{-}s5}$

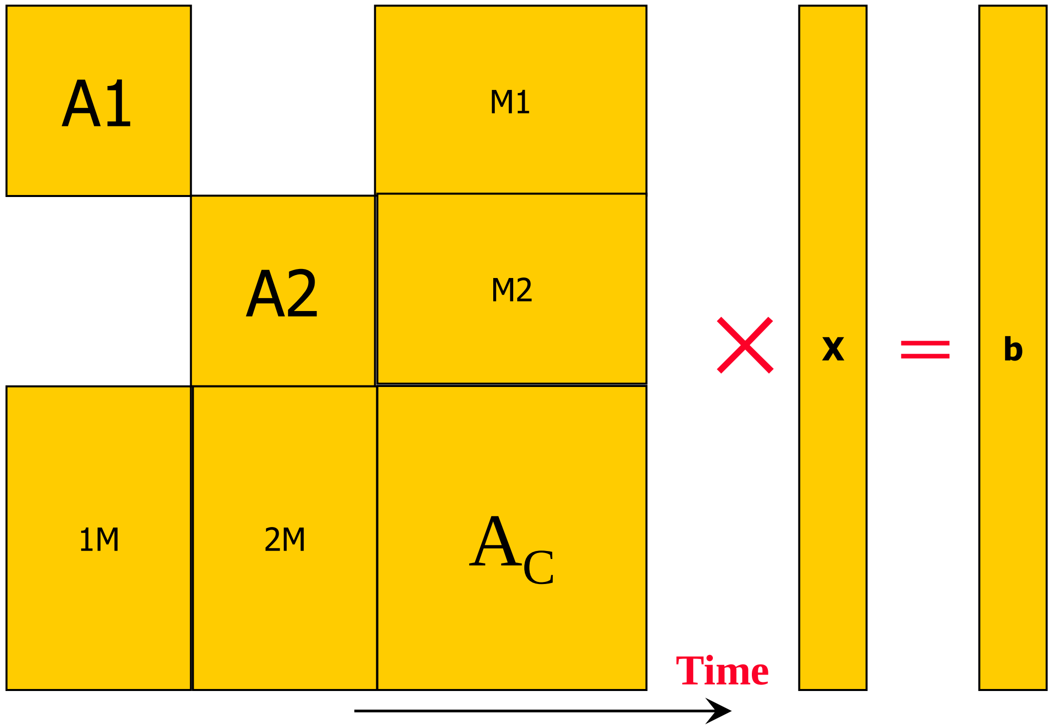
Proc 2: Compute X2 $T_{p2\text{-}s5}$

例子

- 从性能角度，你希望是怎样的并行执行过程？



性能问题



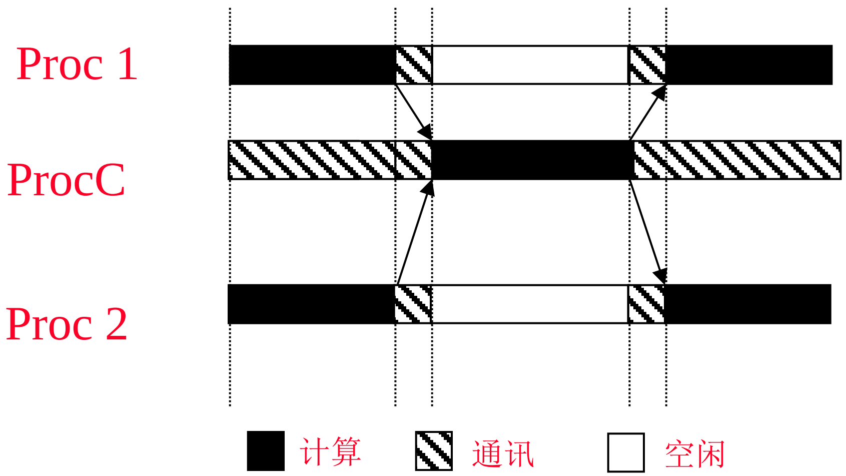
Step 1:
- Proc 1: Compute T1 / Tb1 5s
- Proc 2: Compute T2 / Tb2 5s

Step 2:
- Communicate to Proc C

Step 3:
- Proc C: Compute bc 10s

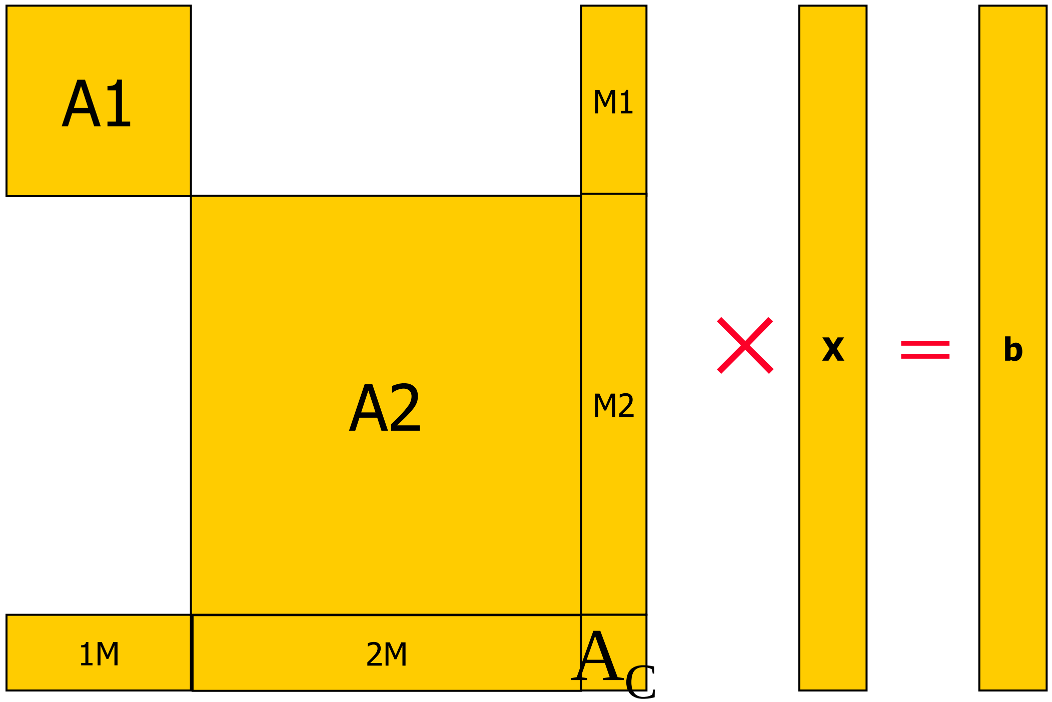
Step 4:
- Communicate to Proc ½

Step 5:
- Proc 1: Compute X1 2s
- Proc 2: Compute X2 2s



For parallel 5+10+2=17s for optimal
For serial 2*5+10+2*2=24s
Only (24-17)/24<30% performance improvement
较大的串行部分是并行程序的性能问题之一

性能问题



Step 1:

- **Proc 1: Compute T1 / Tb1 5s**
- **Proc 2: Compute T2 / Tb2 15s**

Step 2:

- **Communicate to Proc C**

Step 3:

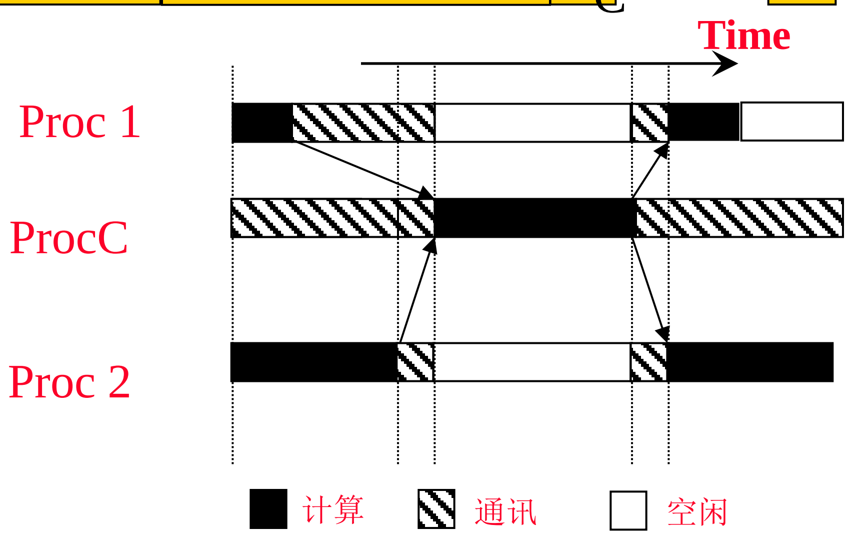
- **Proc C: Compute bc 1s**

Step 4:

- **Communicate to Proc ½**

Step 5:

- **Proc 1: Compute X1 2s**
- **Proc 2: Compute X2 5s**



For parallel 15+1+5=21s for optimal

For serial 15+5+1+2+5=28s

Only (28-21)/28=25% performance improvement

任务划分导致的负载不均衡是并行程序性能问题的重要来源

Summary: Performance issues in Parallel Applications

The primary sources of inefficiency in parallel codes

1. DO NOT have **enough parallelism**
2. Too much parallelism **overhead**
    - Thread creation, **synchronization, communication**
3. **Load imbalance – inefficient task scheduling**
    - Different amounts of work across processors
        - Computation and communication
    - Different speeds (or available resources) for the processors
        - Possibly due to load on the machine
4. **Poor single processor performance**
    - *Typically in the memory system*

Principles of Parallel Computing

- Speedup, efficiency, and Amdahl's Law
- Finding and exploiting parallelism
- Finding and exploiting data locality
- **Load balancing (task scheduling)**
- **Decrease parallel overhead**

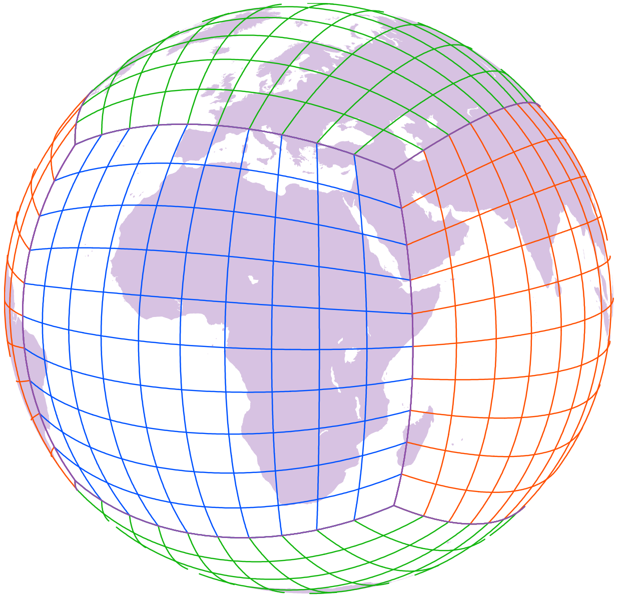
All of these things make parallel programming more difficult than sequential programming.

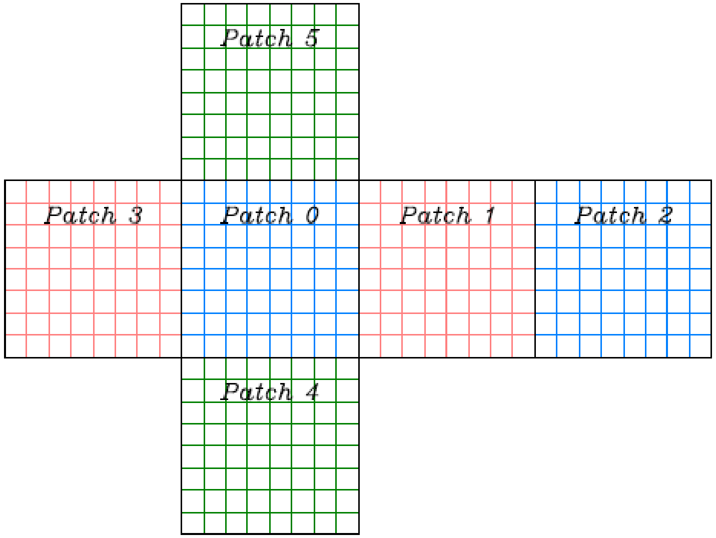
Solving the problem in Parallel Overhead when multiprocessing

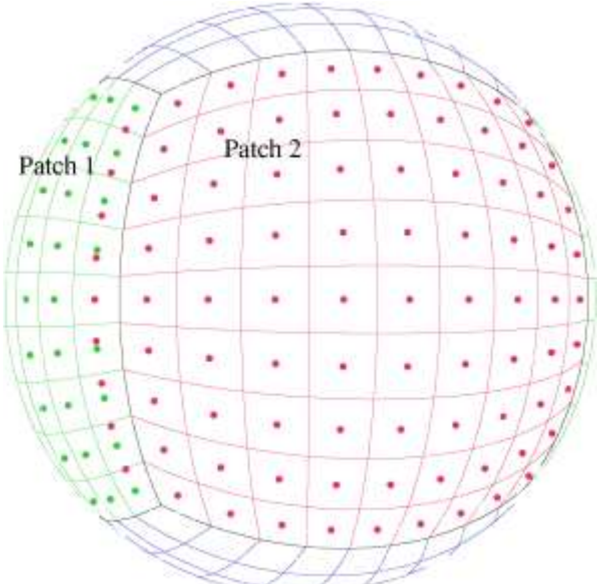
- 静态的进程创建对整体性能影响有限

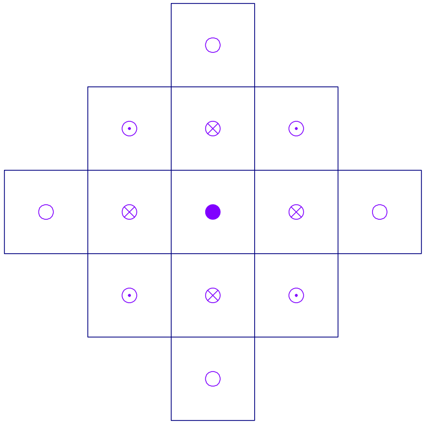
- 核心问题在于解决通信开销问题
  - 通过增加各个进程的计算比重降低通信开销比例
    - 粒度
      - 计算与通信比率
      - Fine-grained:分解成很多小的任务，适合SMP、多核
      - Coarse-grained:分解成不多的大的任务，适合Cluster
    - 粒度选择需要与机器模型的匹配
  - 聚合消息减少通信开销
    - 消息聚合隐藏消息延迟
  - 计算和通信重叠
  - 在高并行度下尽可能减小高开销的群集通信
    - **慎用Scatterv和Gatherv(MPICH) -- 与MPI的实现相关**

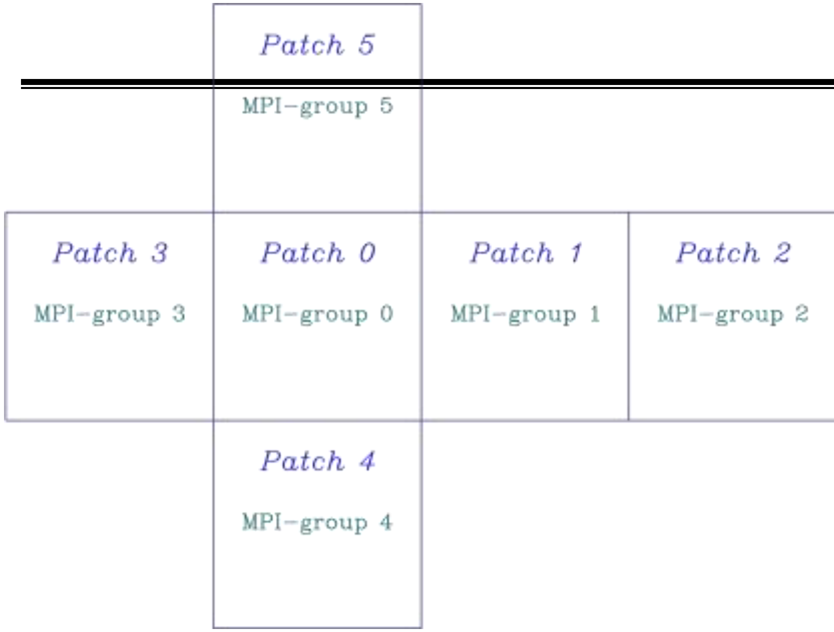
计算通信重叠

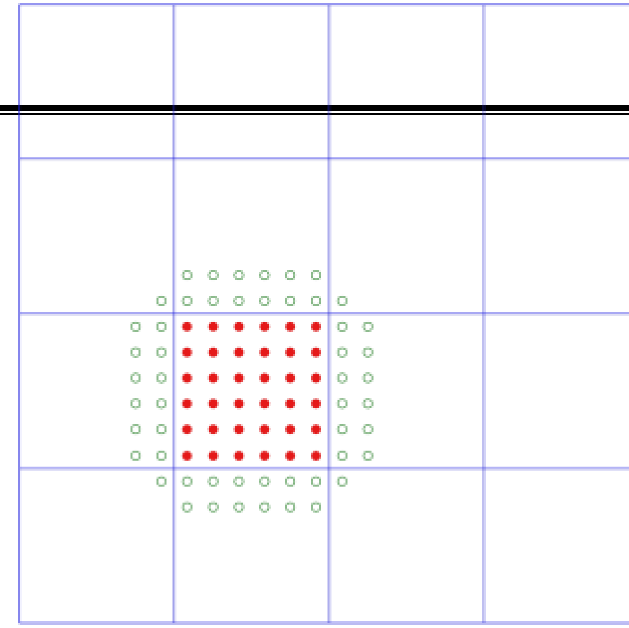


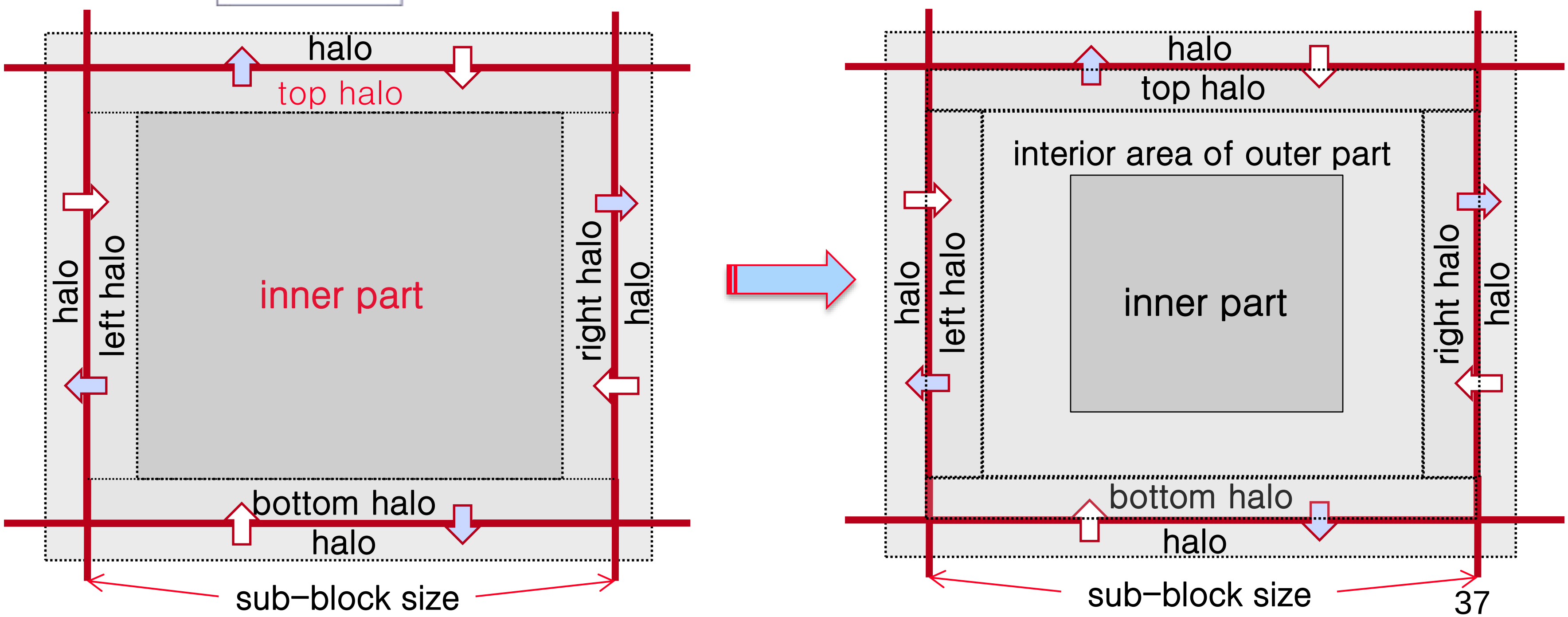


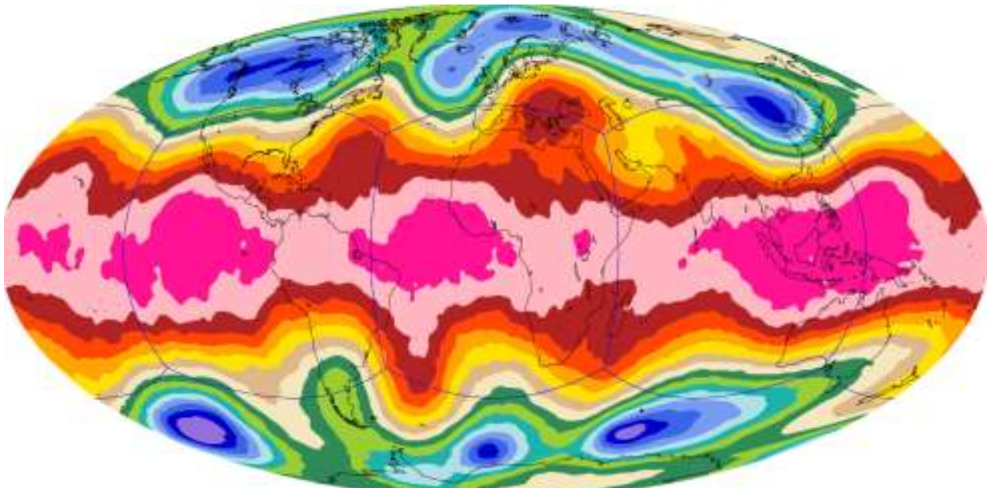




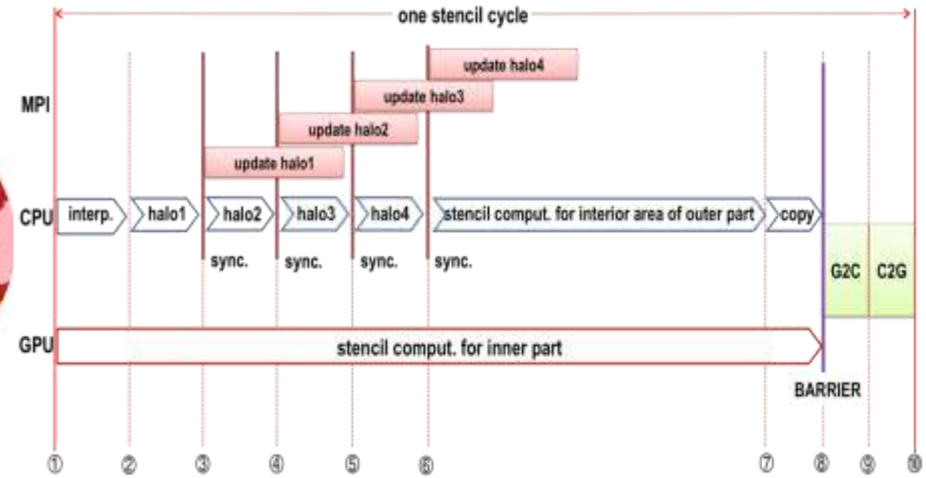

Patch 5
MPI−group 5
Patch 3
MPI−group 3
Patch 0
MPI−group 0
Patch 1
MPI−group 1
Patch 2
MPI−group 2
Patch 4
MPI−group 4



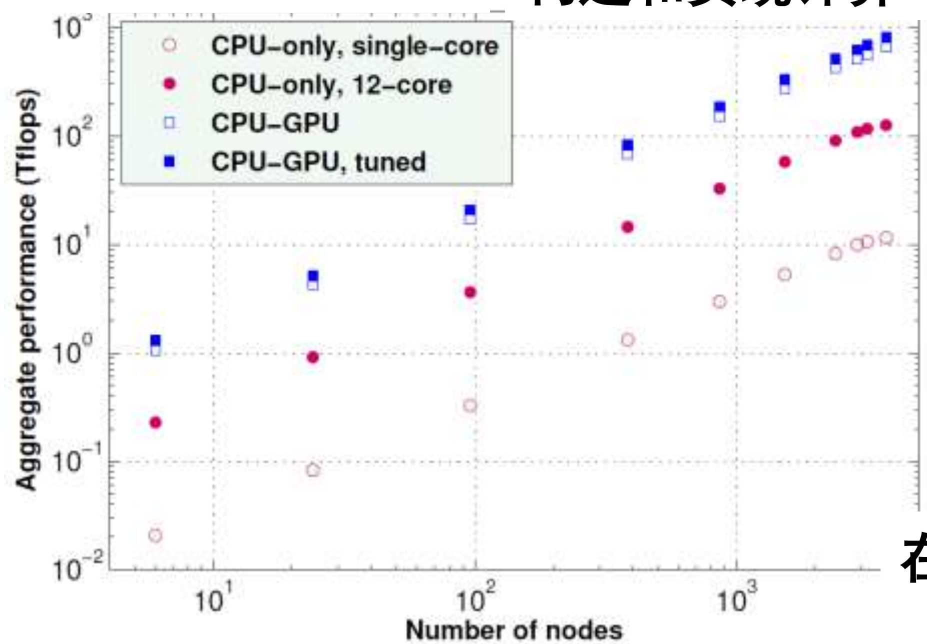

halo
top halo
left halo
halo
inner part
right halo
halo
bottom halo
halo
sub−block size
halo
top halo
interior area of outer part
halo
left halo
inner part
right halo
halo
bottom halo
halo
sub−block size



问题：天河1A上浅水波方程求解优化



构造和实现计算与通讯并发执行的算法



在天河1号上的运行结果

Solving the problem in Parallel Overhead when multithreading

- Thread Creation overhead
  - Overhead increases rapidly as the number of active threads increases
  - Solution: **Use of re-usable threads and thread pools**
    - ➢ Amortizes the cost of thread creation
    - ➢ Keeps number of active threads relatively constant
- Locality
  - Allocate on stack or use thread local storage to release Heap contention
  - Replicate data copies for use by multi-threads
  - False sharing can degrade performance, so organize data efficiently
- Data race overhead
  - Use as less as syn. (explicit and implicit)
  - Keep few active threads to access data area
  - Decrease the size of critical sections
  - Atomic updates versus critical sections
    - Some global data updates can use atomic operations
    - Use atomic updates whenever possible

Task scheduling is an interesting topic, for example

以前的一个例子

$$\dot{X} = f(X,V) = AX + Bu(X,V)$$
$$0 = I - Y(X) * V$$

Computational Model

- Differential Algebra Equations

▪ State equations are easy to parallel

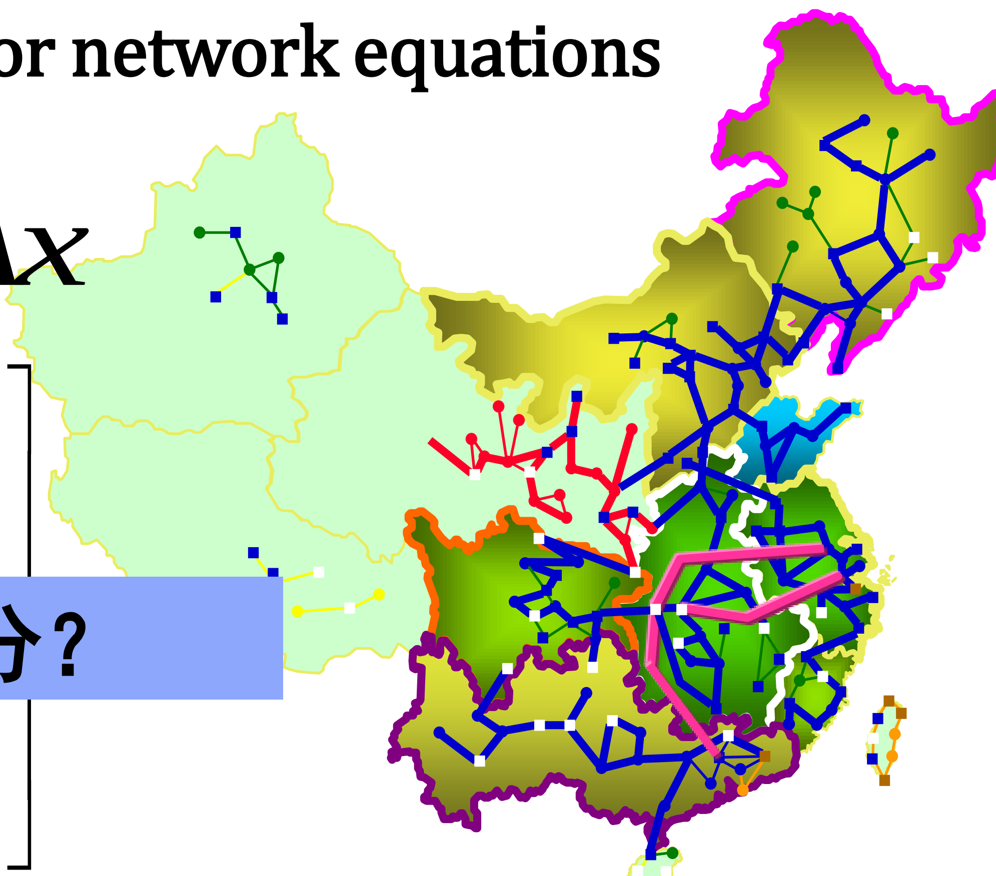
• Spatial parallel algorithm for network equations

$$b = Ax$$

Partition scheme

$$\begin{bmatrix} b_1 \\ \vdots \\ b_p \\ b_s \end{bmatrix} = \begin{bmatrix} A_1 & & & A_{1s} \\ & \ddots & & \vdots \\ & & A_p & A_{ps} \\ A_{s1} & \cdots & A_{sp} & A_s \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_p \\ x_s \end{bmatrix}$$

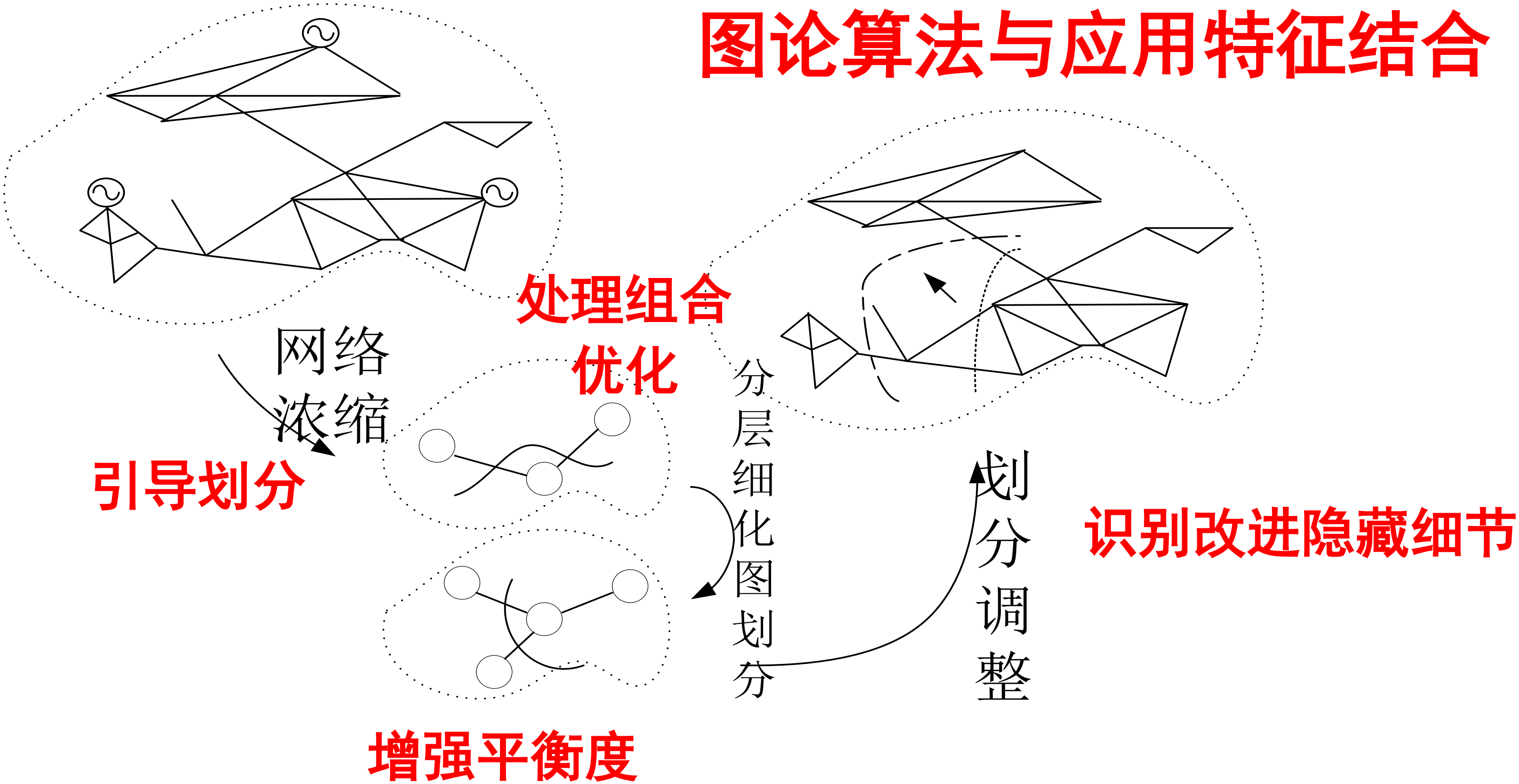
如何进行任务划分？



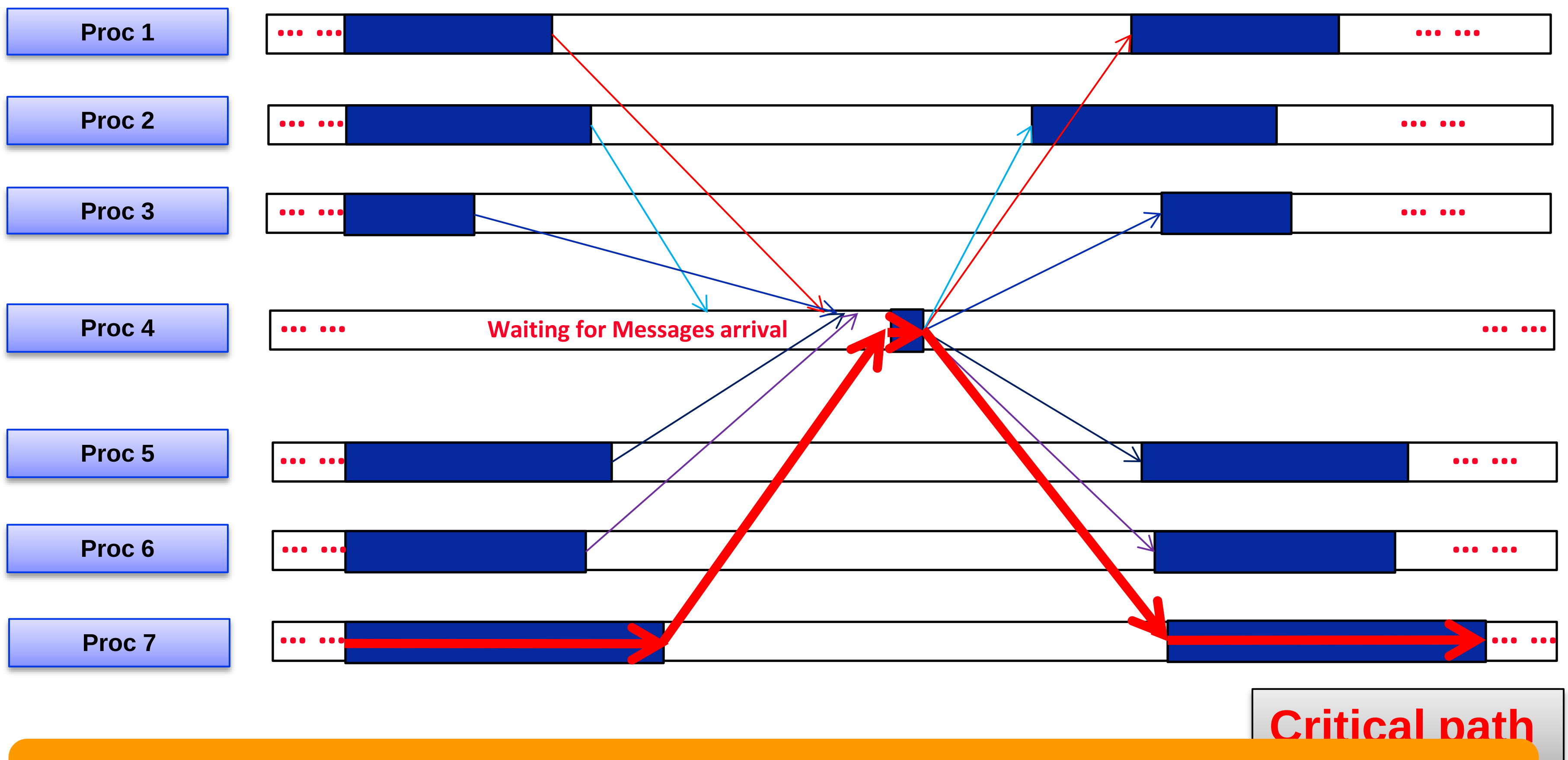
任务划分

- 任务划分
  - 低通信 — 降低固有串行部分比例
  - 高负载均衡—分区计算均衡

图论算法与应用特征结合



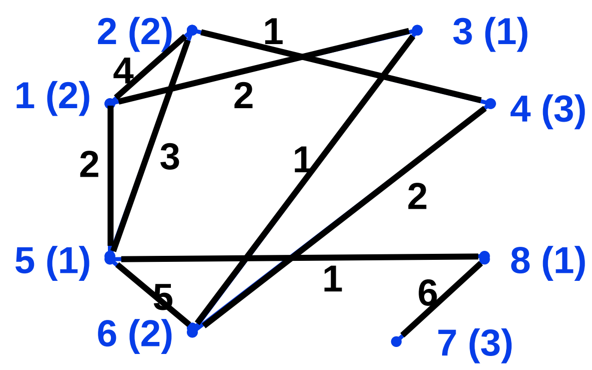
Critical Path



The critical path is the longest execution flow

Definition of Graph Partitioning from Berkeley CS267

- Given a graph $G = (N, E, W_N, W_E)$
  - N = nodes (or vertices),
  - W_N = node weights
  - E = edges
  - W_E = edge weights



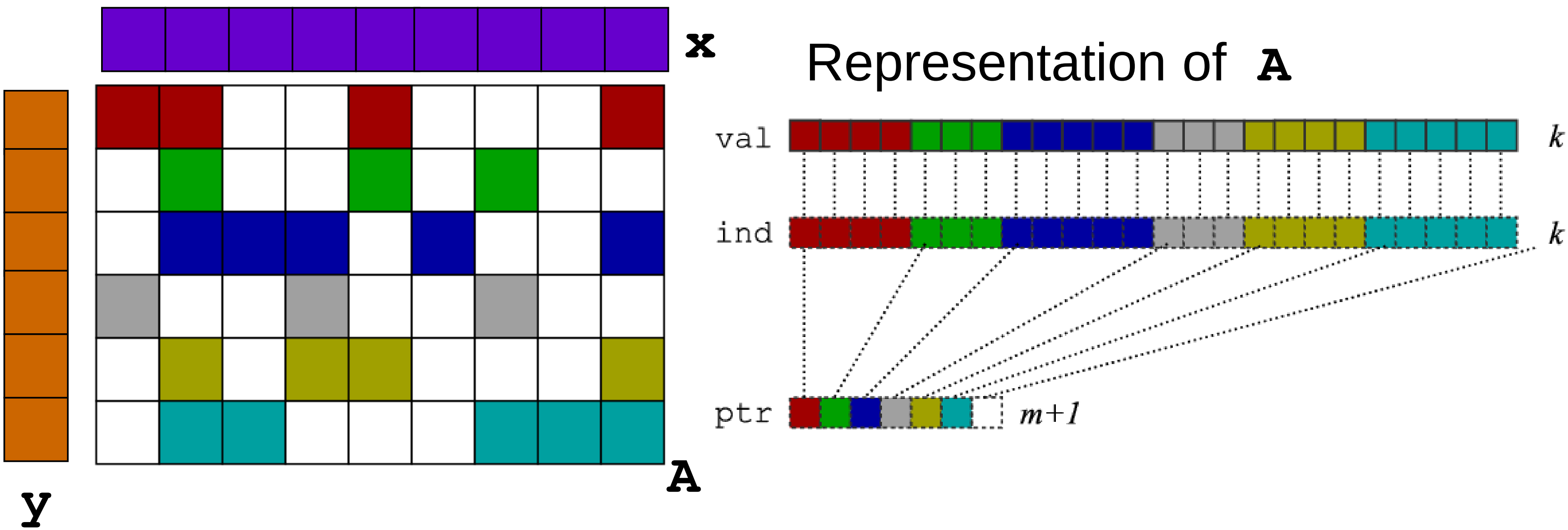
- Ex: N = {tasks}, W_N = {task costs}, edge (j,k) in E means task j sends $W_E(j,k)$ words to task k
- Choose a partition $N = N_1 \cup N_2 \cup \ldots \cup N_P$ such that
  - The sum of the node weights in each N_j is "about the same"
  - The sum of all edge weights of edges connecting all different pairs N_j and N_k is minimized
- Ex: balance the work load, while minimizing communication
- Special case of $N = N_1 \cup N_2$: Graph Bisection

Some Applications

- Telephone network design
  - Original application, algorithm due to Kernighan
- Load Balancing while Minimizing Communication
- Sparse Matrix times Vector Multiplication
  - Solving PDEs
  - $N = \{1,\ldots,n\}$, (j,k) in E if $A(j,k)$ nonzero,
  - $W_N(j) = \#\text{nonzeros in row } j$, $W_E(j,k) = 1$
- VLSI Layout
  - $N = \{\text{units on chip}\}$, $E = \{\text{wires}\}$, $W_E(j,k) = \text{wire length}$
- Sparse Gaussian Elimination
  - Used to reorder rows and columns to increase parallelism, and to decrease "fill-in"
- Data mining and clustering
- Physical Mapping of DNA
- Image Segmentation

SpMV in Compressed Sparse Row (CSR) Format

CSR format is one of many possibilities

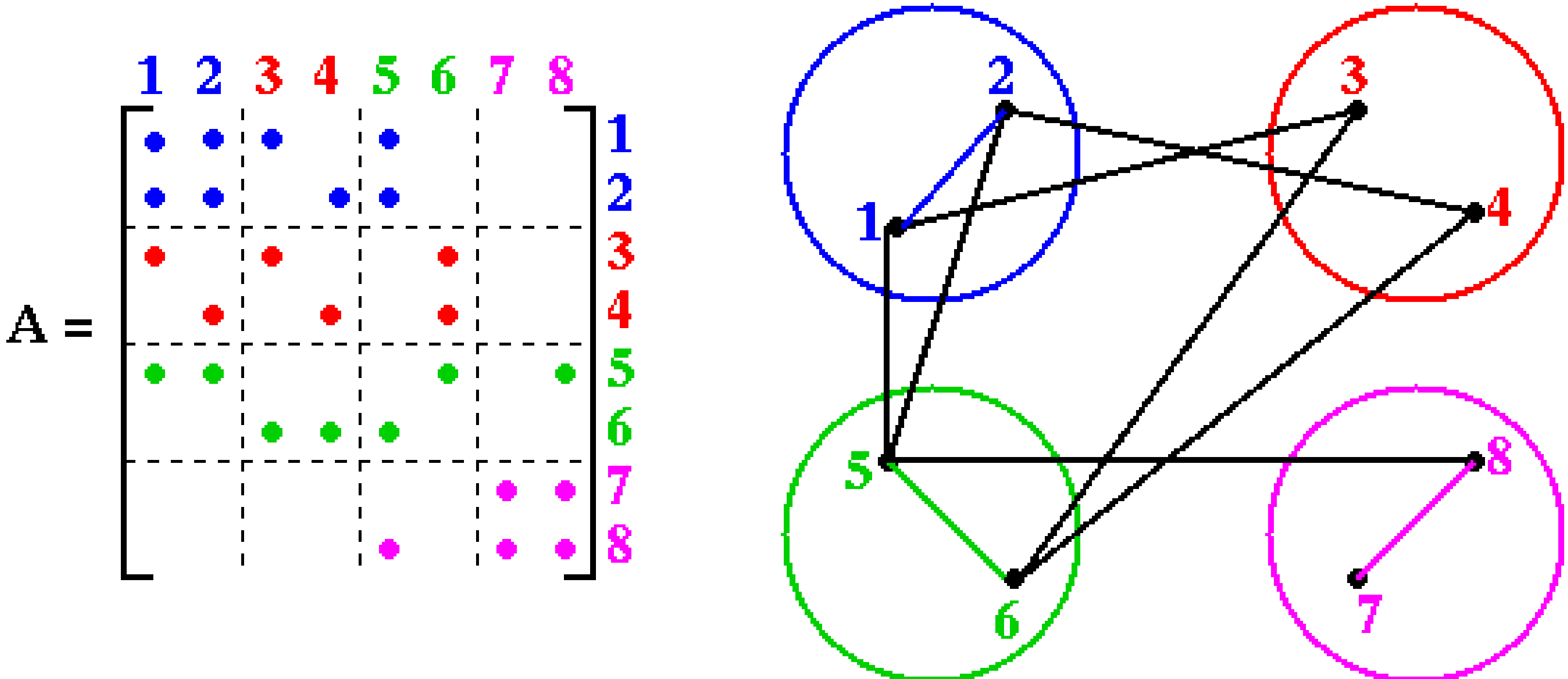


Matrix-vector multiply kernel: $y_{(i)} \leftarrow y_{(i)} + A_{(i,j)} \times x_{(j)}$

```
for each row i
  for k=ptr[i] to ptr[i+1] do
       y[i] = y[i] + val[k]*x[ind[k]]
```

Sparse Matrix Vector Multiplication y = y +A*x

Partitioning a Sparse Symmetric Matrix

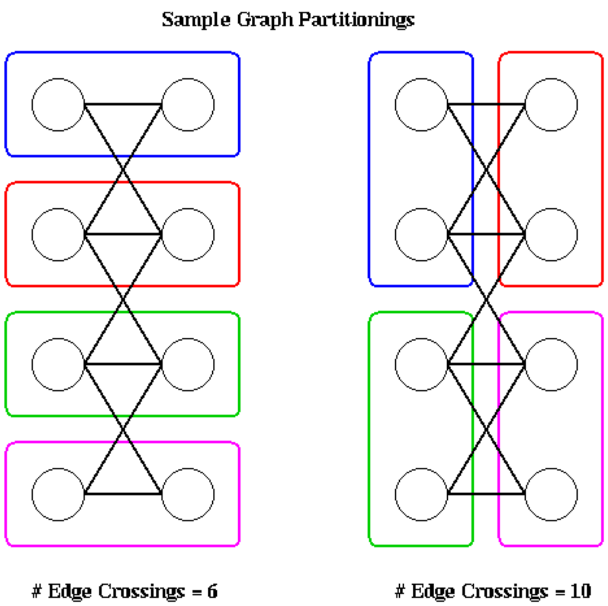



```
… declare A_local, A_remote(1:num_procs), x_local, x_remote, y_local
y_local = y_local + A_local * x_local
for all procs P that need part of x_local
        send(needed part of x_local, P)
for all procs P owning needed part of x_remote
        receive(x_remote, P)
        y_local = y_local + A_remote(P)*x_remote
```

Cost of Graph Partitioning

- Many possible partitionings to search
- Just to divide in 2 parts there are:

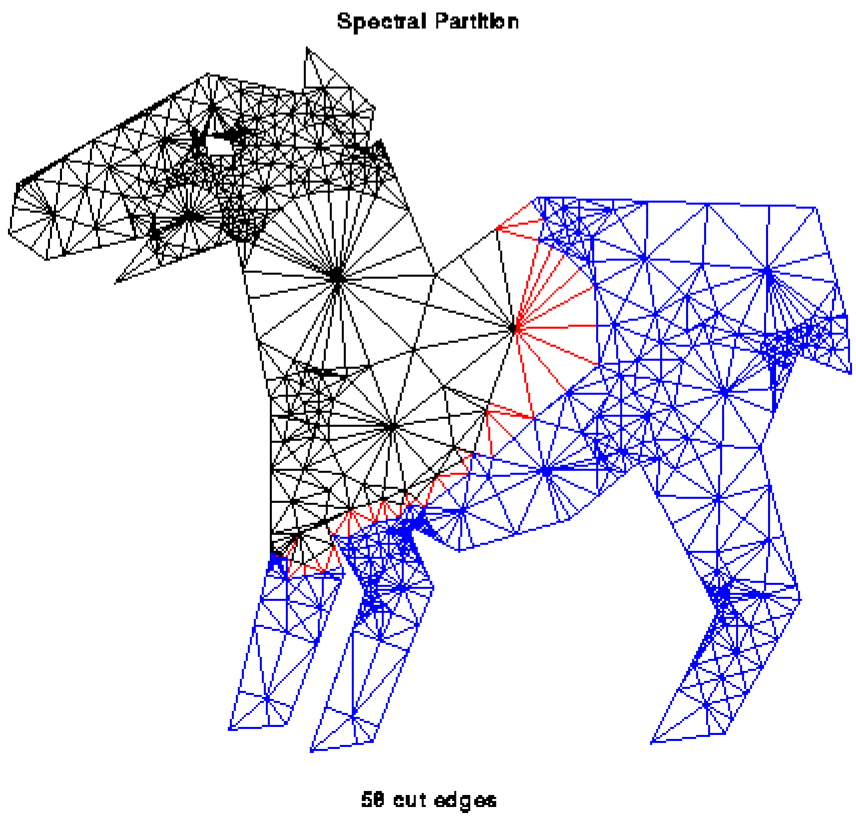
  n choose n/2



- **Choosing optimal partitioning is NP-complete**
  - **(NP-complete = we can prove it is a hard as other well-known hard problems in a class Nondeterministic Polynomial time)**
  - **Only known exact algorithms have cost = exponential(n)**
- **We need good heuristics**

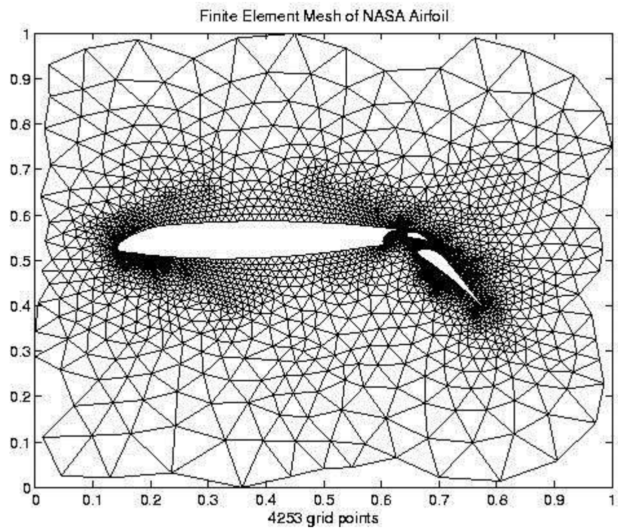
First Heuristic: Repeated Graph Bisection

- To partition N into 2^k parts
  - bisect graph recursively k times
- Henceforth discuss mostly graph bisection



Overview of Bisection Heuristics

- Partitioning with Nodal Coordinates
  - Each node has x,y,z coordinates → partition space



- Partitioning … ates
  - E.g., Sparse matrix of Web documents
    - A(j,k) = # times keyword j appears in URL k
- Multilevel acceleration (BIG IDEA)
  - Approximate problem by "coarse graph," do so recursively

Available Implementations

- Multilevel Kernighan/Lin
  - METIS (www.cs.umn.edu/~metis)
  - ParMETIS - parallel version
- Multilevel Spectral Bisection
  - S. Barnard and H. Simon, "A fast multilevel implementation of recursive spectral bisection …", Proc. 6th SIAM Conf. On Parallel Processing, 1993
  - Chaco (www.cs.sandia.gov/CRF/papers_chaco.html)
- Hybrids possible
  - Ex: Using Kernighan/Lin to improve a partition from spectral bisection
- Recent package, collection of techniques
  - Zoltan (www.cs.sandia.gov/Zoltan)

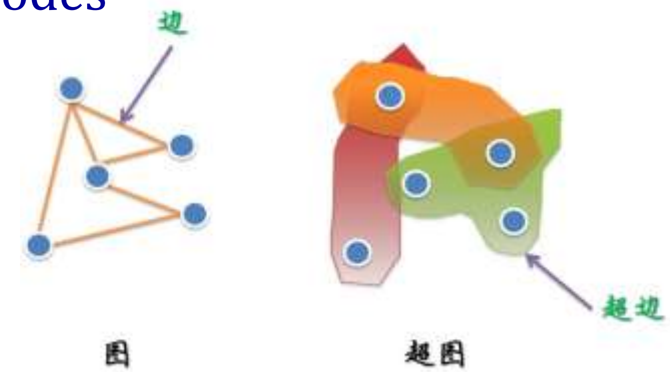
Comparison of methods

- Metrics
  - Speed of partitioning
  - Number of edge cuts
  - Other application dependent metrics

- Summary
  - No one method best
  - Multi-level Kernighan/Lin fastest by far, comparable to Spectral in the number of edge cuts
    - www-users.cs.umn.edu/~karypis/metis/publications/main.html
    - see publications KK95a and KK95b
  - Spectral give much better cuts for some applications
    - Ex: image segmentation
    - See "Normalized Cuts and Image Segmentation" by J. Malik, J. Shi

Beyond Simple Graph Partitioning

- Undirected graphs model symmetric matrices, not unsymmetric ones
- More general graph models include:
  - Hypergraph: nodes are computation, edges are communication, but connected to a set (>= 2) of nodes
    - HMETIS package

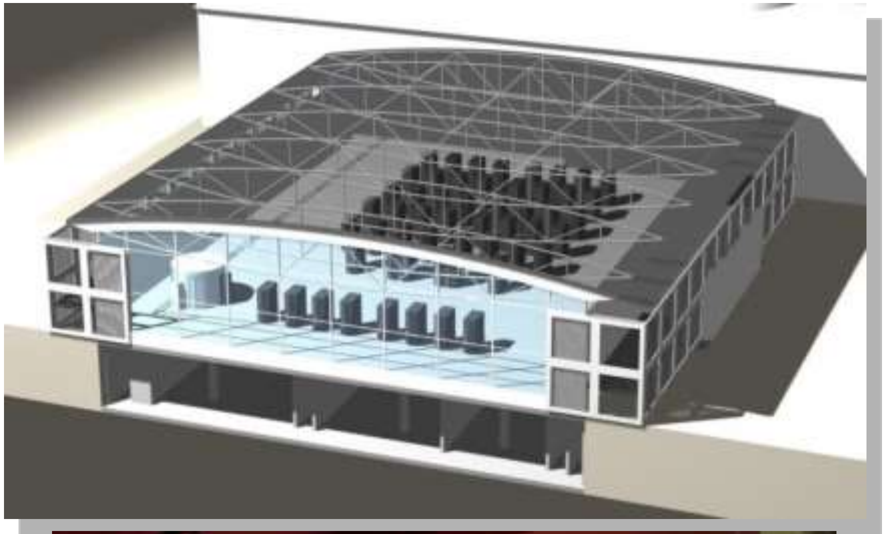


  - Multi-object, Multi-Constraint model: use when single structure may involve multiple computations with differing costs
- For more see Bruce Hendrickson's web page
  - www.cs.sandia.gov/~bahendr/partitioning.html
  - "Load Balancing Myths, Fictions & Legends"

performance tools for parallel programming

Motivation for performance tools [from Berkeley CS267]

- Performance analysis is important
  - For HPC: computer systems are large investments
    - Procurement: O($40 Mio)
    - Operational costs: ~$5 Mio per year
    - Power: 1 MWyear ~$1 Mio
  - Goals:
    - Solve **larger** problems (new science)
    - Solve problems **faster** (turn-around time)
    - Improve **error** bounds on solutions (confidence)



Intel® VTune™ Amplifier XE
Performance Profiler

Where is my application…

Spending Time?

Function - Call Stack	CPU Time
algorithm_2	3.560s
do_xform	3.560s
algorithm_1	1.412s
BaseThreadInitThı	0.000s

- Focus tuning on functions taking time
- See call stacks
- See time on source

Wasting Time?

Line		MEM_LOAD... LLC_MISS
475	float rx, ry, rz =	
476	float param1 = (AA	30,000
477	float param2 = (AA	
478	bool neg = (rz < C	

- See cache misses on your source
- See functions sorted by # of cache misses

Waiting Too Long?

Wait Time (Idle / Poor / Ok / Ideal)	Wait Count
176.504s	18,277
84.681s	5,499
84.612s	5,489

- See locks by wait time
- Red/Green for CPU utilization during wait

- Windows & Linux
- Low overhead
- No special recompiles

- ✓ 热点分析
- ✓ 并行度分析
- ✓ 锁和等待分析
- ✓ 对比分析

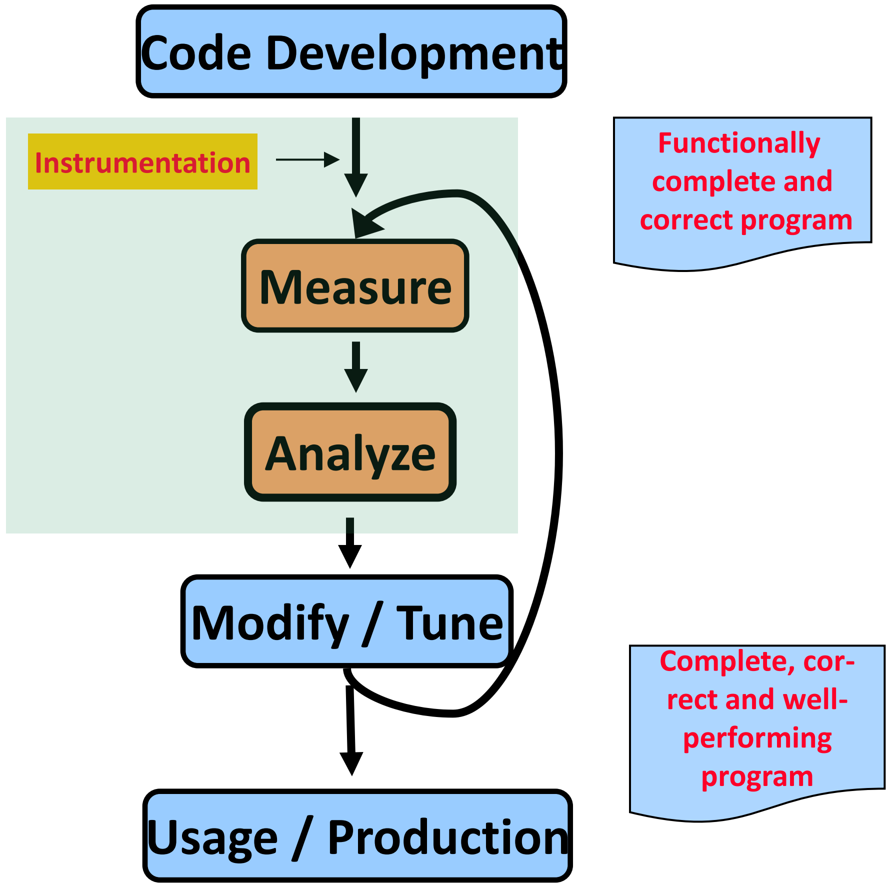
We improved the performance of the latest run 3 fold. We wouldn't have found the problem without something like Intel® VTune™ Amplifier XE.

Claire Cates

Principal Developer, SAS Institute Inc.

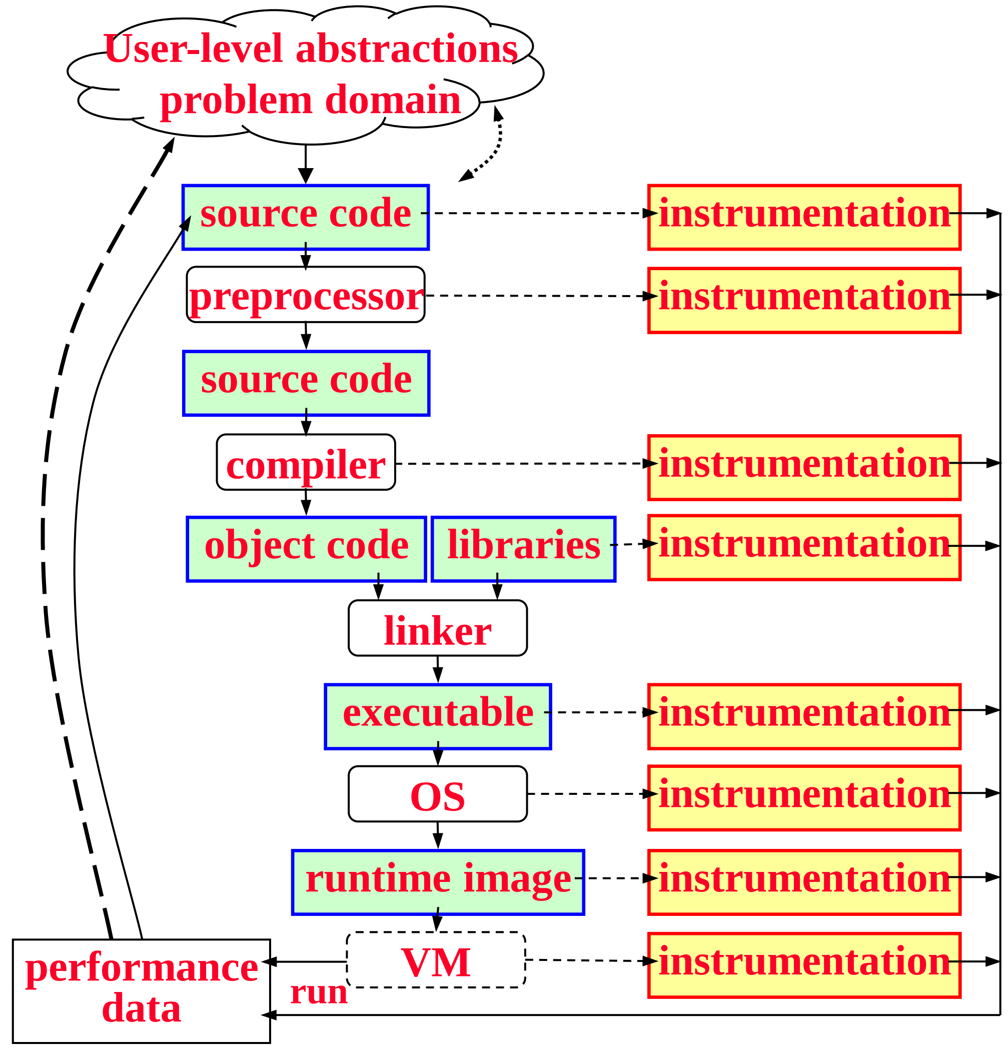
Concepts and Definitions

- The typical performance optimization cycle



Instrumentation

- Instrumentation := adding measurement probes to the code in order to observe its execution
- Can be done on several levels and ddifferent techniques for different levels
- Different overheads and levels of accuracy with each technique
- No application instrumentation needed: run in a simulator. E.g., Valgrind, SIMICS, etc. but simulation speed is an issue

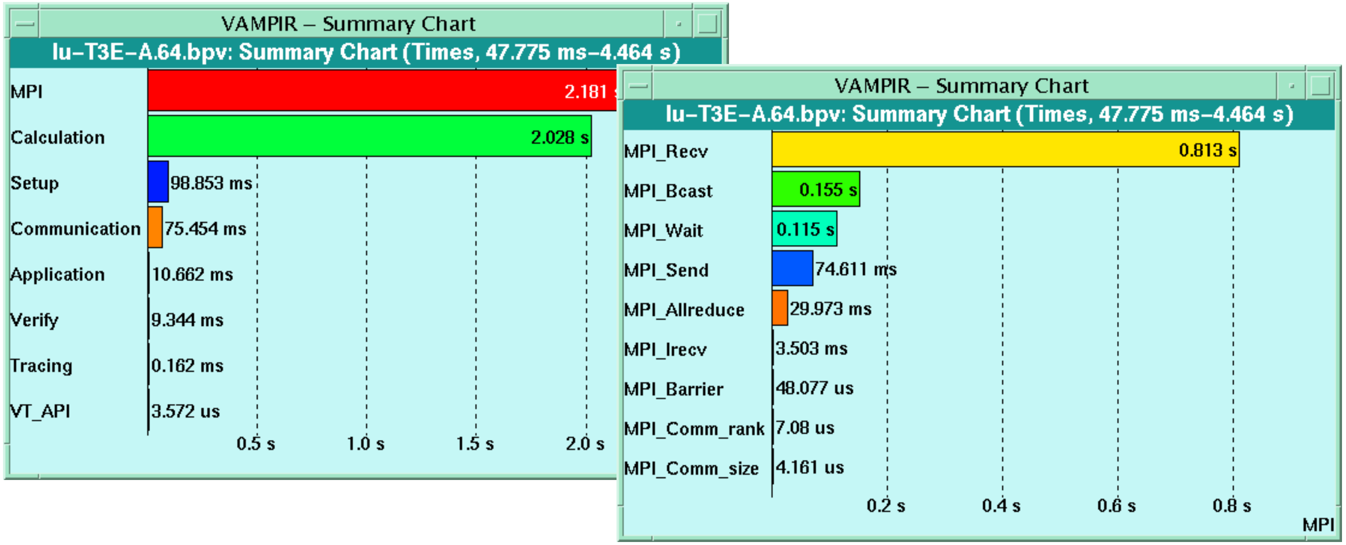


Vampir – Trace Visualization

（ http://www.vampir.eu/ ）

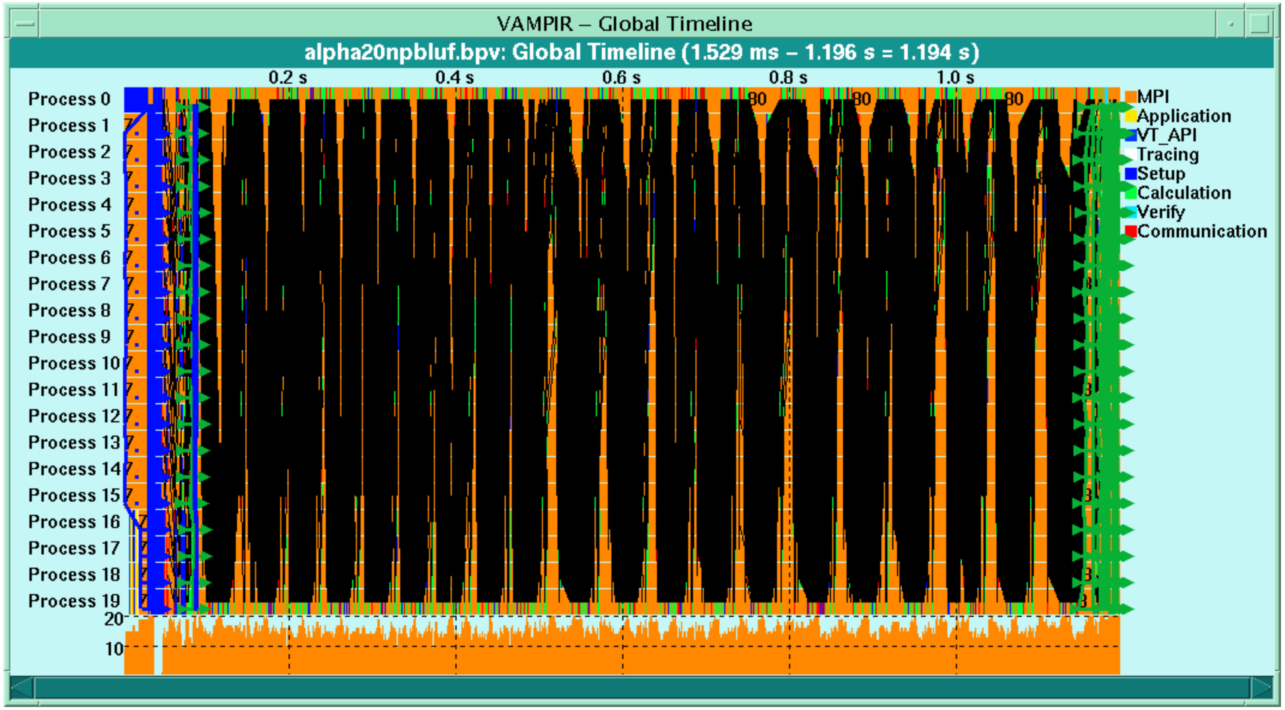
Intel Trace Collector and Analyzer

Vampir overview statistics



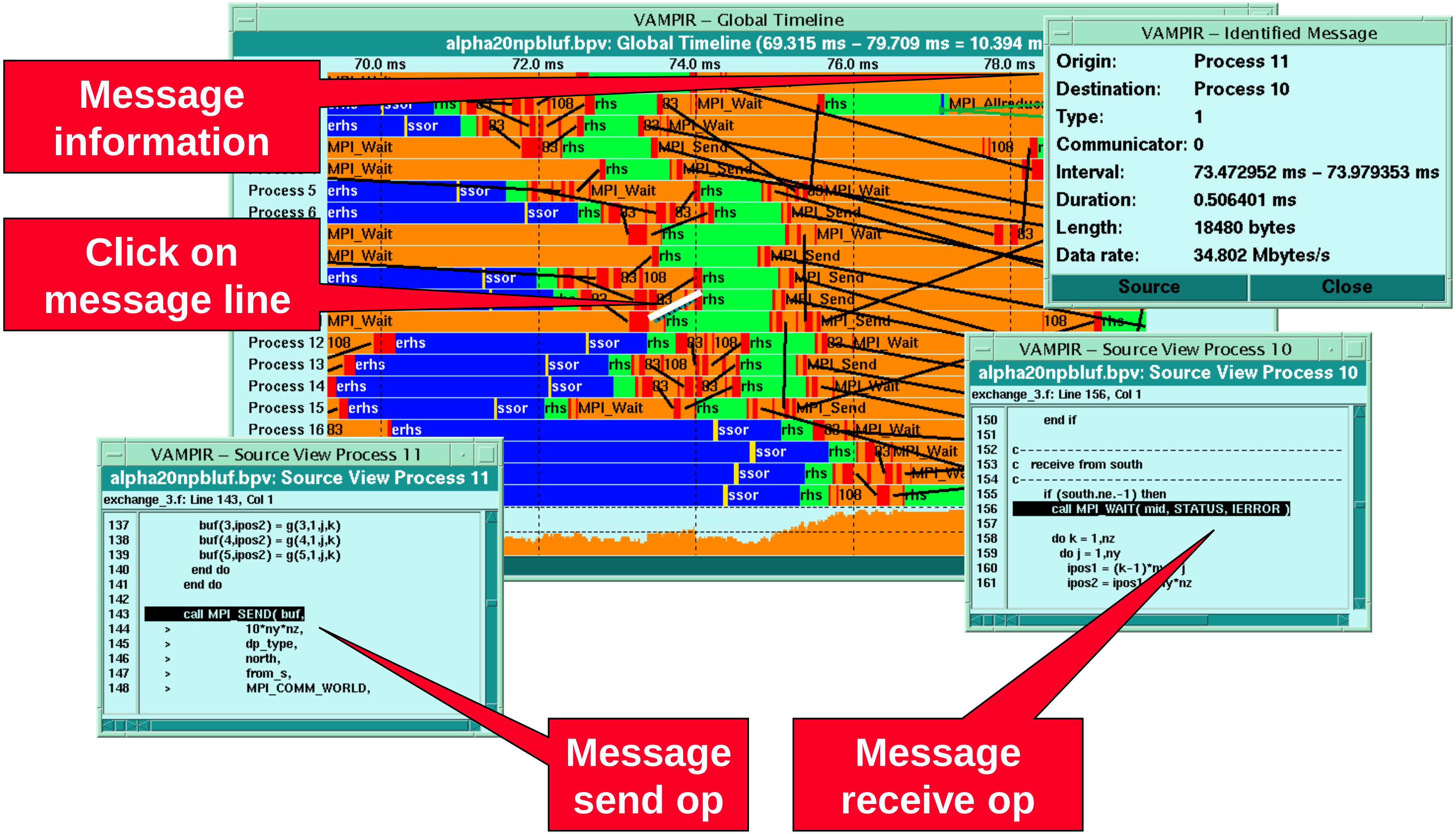
- Aggregated profiling information
  - Execution time
  - Number of calls
- This profiling information is computed from the trace
  - Change the selection in main timeline window
- Inclusive or exclusive of called routines

Timeline display

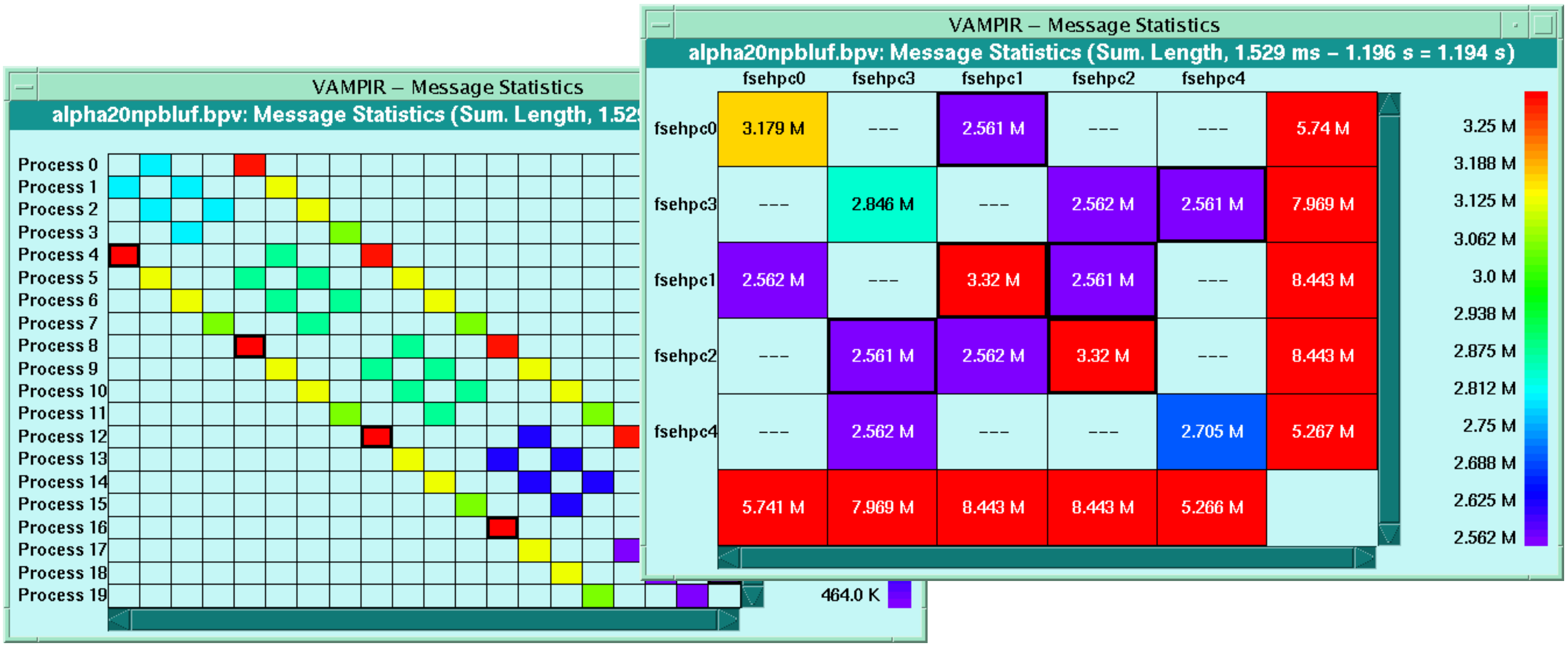


- To zoom, mark region with the mouse

Timeline display – message details

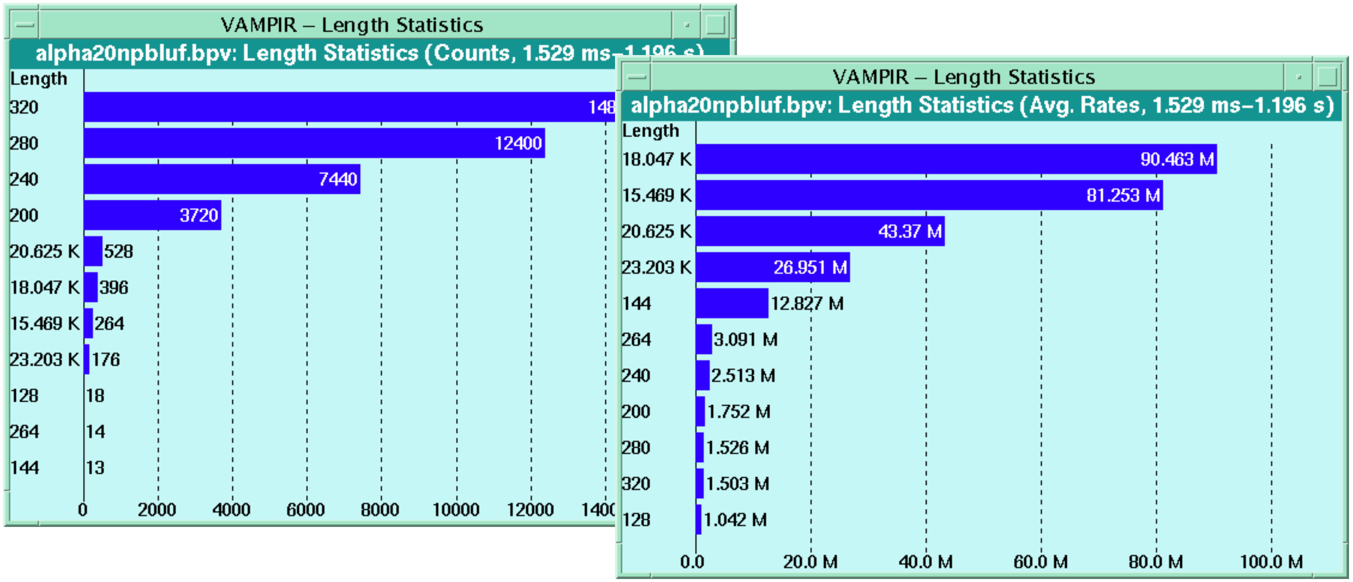


Communication statistics



- Message statistics for each process/node pair:
    - Byte and message count
    - min/max/avg message length, bandwidth

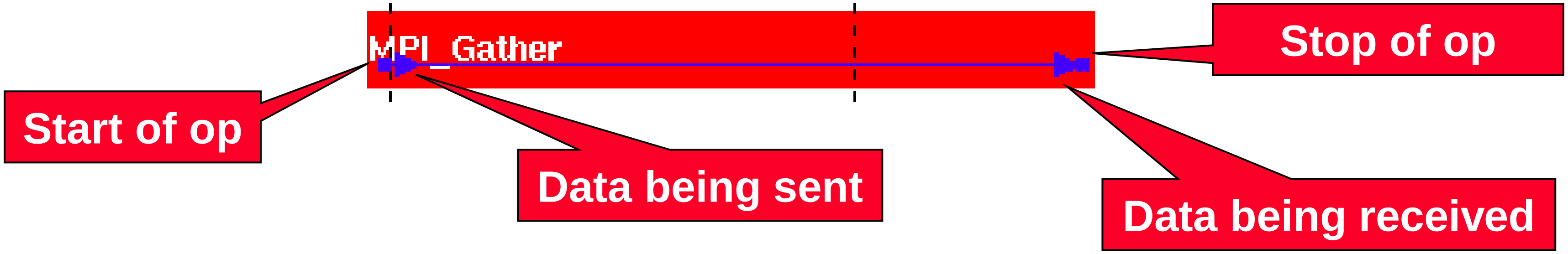
Message histograms



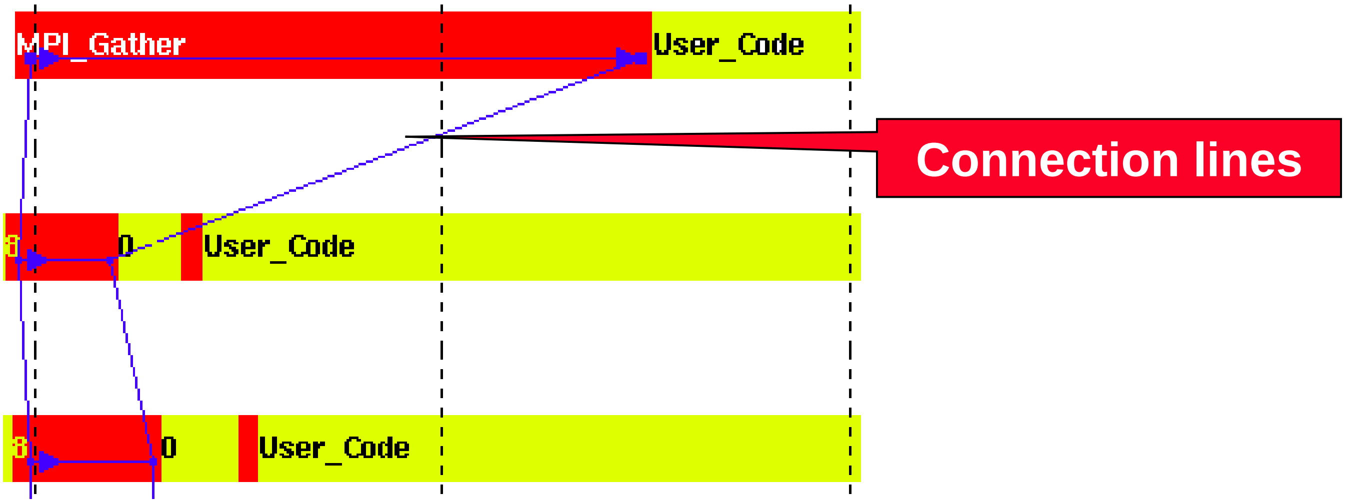
- Message statistics by length, tag or communicator
  - Byte and message count
  - Min/max/avg bandwidth

Collective operations

- For each process: mark operation locally

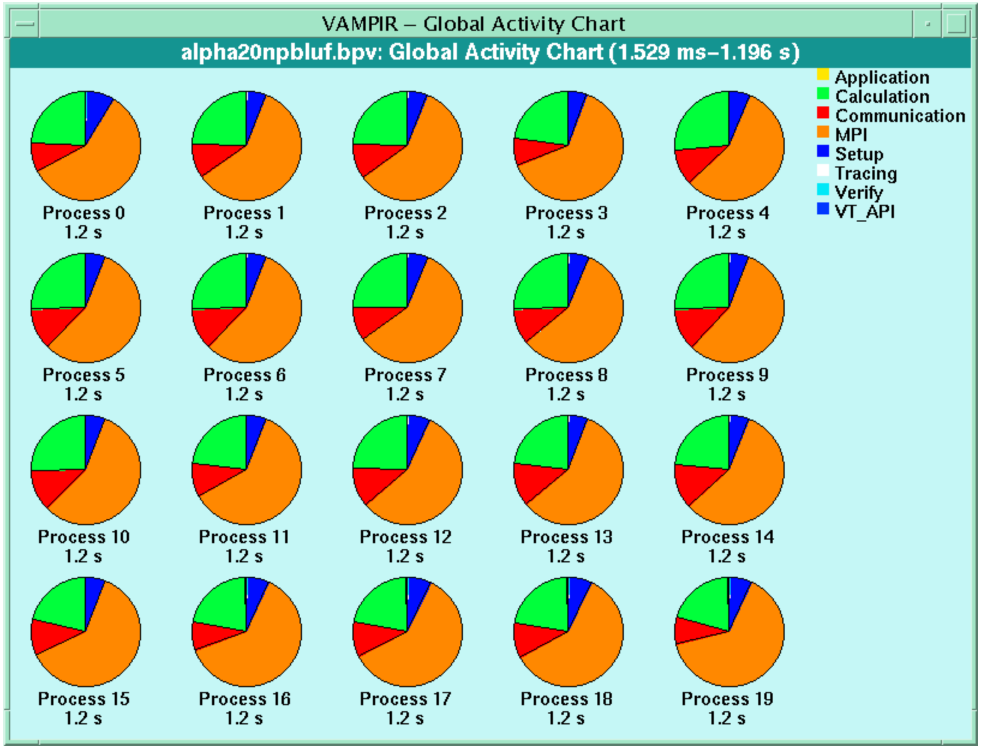


- Connect start/stop points by lines



Activity chart

- Profiling information for all processes



Summary

- Performance montioring concepts
  - Instrument, measure, analyze
  - Profiling/tracing, sampling, direct measurment

- Tools
  - PAPI, ompP, and IPM as examples
  - Vendor tools: Cray PAT, Sun Studio, Intel Thread Profiler, Vtune, PTU,…
  - Portable tools: TAU, Perfsuite, Paradyn, HPCToolkit, Kojak, Scalasca, Vampir, oprofile, gprof, …

Documentation, Manuals, User Guides

✓PAPI

- http://icl.cs.utk.edu/papi/

✓ompP

- http://www.ompp-tool.com

✓IPM

- http://ipm-hpc.sourceforge.net/

✓TAU

- http://www.cs.uoregon.edu/research/tau/

✓VAMPIR / INTEL Trace Collector and Analyzor

- http://www.vampir-ng.de/ http://software.intel.com/en-us/intel-trace-analyzer

✓Scalasca

- http://www.scalasca.org