

并行计算

Parallel Computing

Spring, 2022

并行计算——结构·算法·编程

- 第一篇 并行计算的基础
 - 第一章 并行计算与并行计算机结构模型
 - 第二章 并行计算机系统互连与基本通信操作
 - 第三章 典型并行计算机系统介绍
 - 第四章 并行计算性能评测

第四章 并行计算性能评测

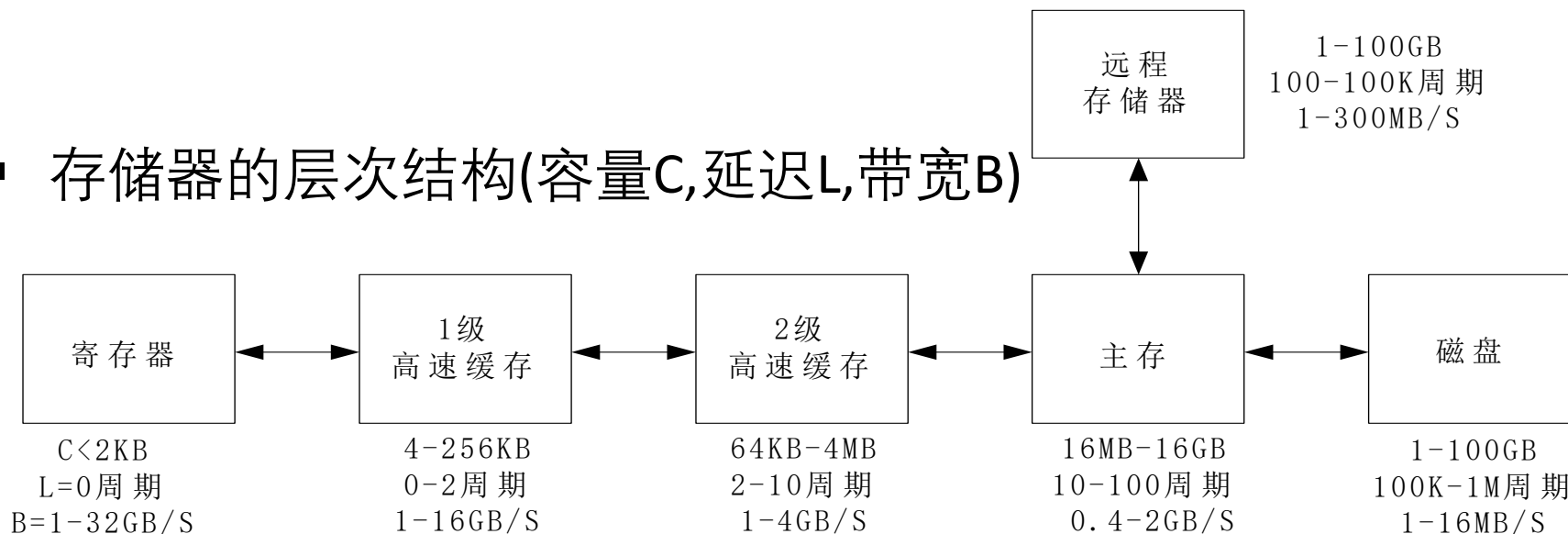
- 4.1 并行机的一些基本性能指标
- 4.2 加速比性能定律
 - 4.2.1 Amdahl定律
 - 4.2.2 Gustafson定律
 - 4.2.3 Sun和Ni定律
- 4.3 可扩展性评测标准
 - 4.3.1 并行计算的可扩展性
 - 4.3.2 等效率度量标准
 - 4.3.3 等速度度量标准
 - 4.3.4 平均延迟度量标准
- 4.4 基准测试程序

CPU的某些基本性能指标

- 工作负载
 - 执行时间
 - 浮点运算数: Flops
 - 指令数目: MIPS
- 无重叠的假定下：并行执行时间包括：
 - T_{comput} 计算时间, T_{paro} 并行开销时间, T_{comm} 相互通信时间
$$T_n = T_{\text{comput}} + T_{\text{paro}} + T_{\text{comm}}$$
 - T_{paro} : 进程管理（如进程生成、结束和切换等），组操作（如进程组的生成与消亡等），进程查询（如询问进程的标志、等级、组标志和组大小等）
 - T_{comm} : 同步（如路障、锁、临界区、事件等），通信（如点到点通信、整体通信），聚合操作（如规约、前缀运算等）

存储器性能

■ 存储器的层次结构(容量C,延迟L,带宽B)



■ 估计存储器的带宽

RISC的加法可在单拍内完成, 假定字长8bytes, 时钟频率100MHz, 则

$$\text{带宽} B = 3 * 8 * 100 * 10^6 \text{ B/s} = 2.4\text{GB/s}$$

并行与通信开销

- 并行和通信开销：相对于计算很大。
PowerPC (每个周期 15ns 执行4flops;
创建一个进程1.4ms, 可执行372000flops)
- 开销的测量：乒--乓方法 (Ping-Pong Scheme) 节点0发送m个字节给节点1; 节点1从节点0接收m个字节后, 立即将消息发回节点0。总的时间除以2, 即可得到点到点通信时间, 也就是执行单一发送或接收操作的时间。
- 可一般化为热土豆法 (Hot-Potato), 也称为救火队法 (Fire-Brigade) 0——1 —— 2 —— ... —— n-1
—— 0

- Ping-Pong Scheme

```
if (my_node_id = 0) then /*发送者*/  
    start_time = second ( )  
        send an m-byte message to node 1  
        receive an m-byte message from node 1  
    end_time = second ( )  
    total_time = end_time - start_time  
    communication_time[i] = total_time/2  
else if (my_node_id = 1) then /*接收者*/  
    receive an m-byte message from node 0  
    send an m-byte message to node 0  
endif
```

并行开销的表达式：点到点通信

- 通信开销 $t(m) = t_0 + m/r_\infty$ m 为消息长度（字节数）

通信启动时间 t_0

渐近带宽 r_∞ ： 传送无限长的消息时的通信速率

半峰值长度 $m_{1/2}$ ： 达到一半渐近带宽所要的消息长度

特定性能 π_0 ： 表示短消息带宽

4个参数只有两个是独立的： $t_0 = m_{1/2} / r_\infty = 1 / \pi_0$

- 以上是由Hockney提出的

并行开销的表达式：整体通信

- 典型的整体通信有：
 - 播送 (Broadcasting)：处理器0发送 m 个字节给所有的 n 个处理器
 - 收集 (Gather)：处理器0接收所有 n 个处理器发来的消息，所以处理器0最终接收了 $m \cdot n$ 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 $m \cdot n$ 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 $m \cdot n^2$ 个字节；
 - 循环移位 (Circular-shift)：处理器 i 发送 m 个字节给处理器 $i+1$ ，处理器 $n-1$ 发送 m 个字节给处理器0，所以通信量为 $m \cdot n$ 个字节。

机器的成本、价格与性/价比

- 机器的成本与价格
- 机器的性能/价格比 Performance/Cost Ratio：系指用单位代价（通常以百万美元表示）所获取的性能（通常以MIPS或MFLOPS表示）
- 利用率（Utilization）：可达到的速度与峰值速度之比

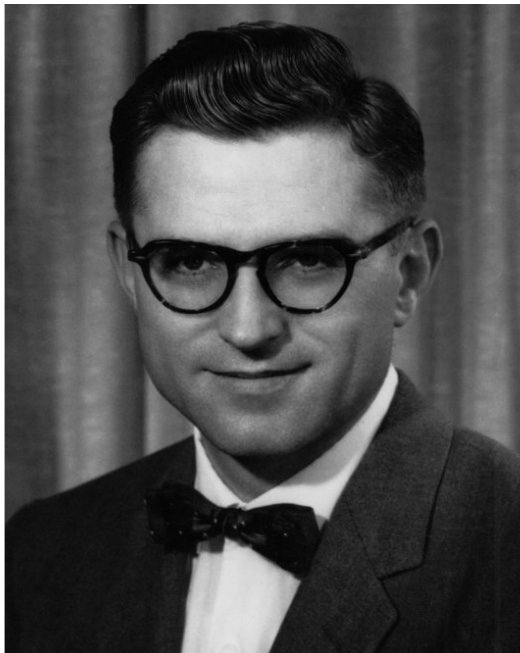
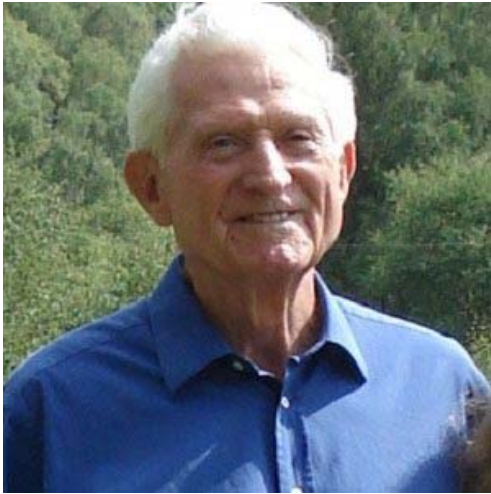
算法级性能评测

- 加速比性能定律
 - 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）的执行速度加快多少倍。
 - Amdahl 定律
 - Gustafson定律
 - Sun Ni定律
- 可扩展性评测标准
 - 等效率度量标准
 - 等速度度量标准
 - 平均延迟度量标准

第四章 并行计算性能评测

- 4.1 并行机的一些基本性能指标
- **4.2 加速比性能定律**
 - 4.2.1 Amdahl定律
 - 4.2.2 Gustafson定律
 - 4.2.3 Sun和Ni定律
- 4.3 可扩放性评测标准
 - 4.3.1 并行计算的可扩放性
 - 4.3.2 等效率度量标准
 - 4.3.3 等速度度量标准
 - 4.3.4 平均延迟度量标准
- 4.4 基准测试程序

Gene Amdahl (1922 — 2015)



Famous for formulating a computer science concept known as Amdahl's Law (1967) and for establishing a major IT company called the Amdahl Corporation, Amdahl is also notable for his work with IBM.

Amdahl 定律 (1)

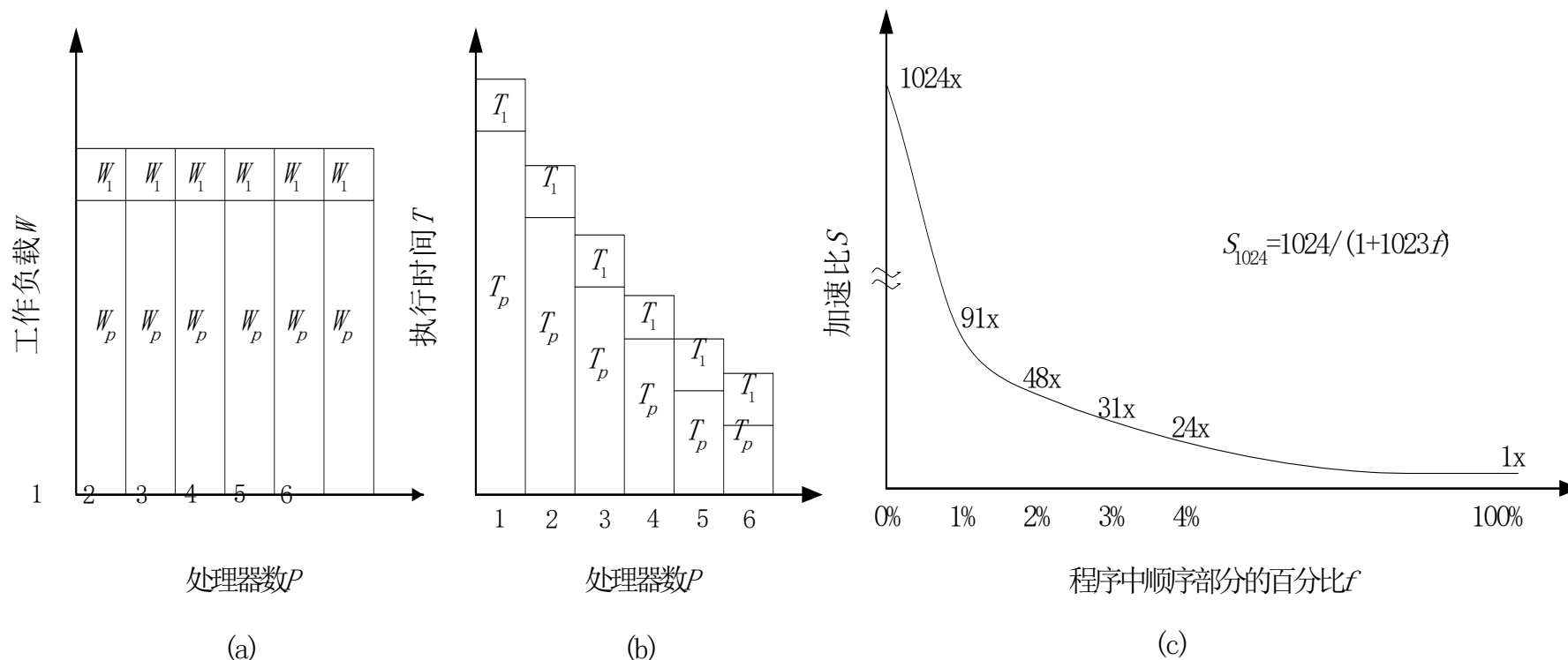
- P : 处理器数;
- W : 问题规模 (计算负载、工作负载, 给定问题的总计算量);
 - W_s : 应用程序中的串行分量, f 是串行分量比例 ($f = W_s/W$, $W_s = W_1$);
 - W_p : 应用程序中可并行化部分, $1-f$ 为并行分量比例;
 - $W_s + W_p = W$;
- $T_s = T_1$: 串行执行时间, T_p : 并行执行时间;
- S : 加速比, E : 效率;
- 出发点: Base on Fixed Problem Size
 - 固定不变的计算负载;
 - 固定的计算负载分布在多个处理器上的,
 - 增加处理器加快执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

Amdahl定律 (2)

- Amdahl's Law 表明:
 - 适用于实时应用问题。当问题的计算负载或规模固定时，我们必须通过增加处理器数目来降低计算时间；
 - 加速比受到算法中串行工作量的限制。
 - Amdahl's law: argument against massively parallel systems
- 公式推导

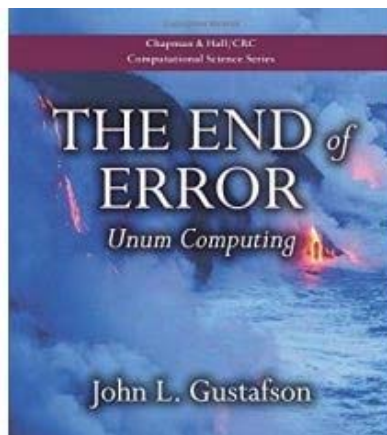
$$T_s = fW + (1-f)W \quad T_p = fW + \frac{(1-f)W}{p}$$
$$S_p = \frac{W}{fW + \frac{(1-f)W}{p}} = \frac{p}{pf + 1 - f} = \frac{1}{\frac{(p-1)f + 1}{p}} \xrightarrow{p \rightarrow \infty} \frac{1}{f}$$

Amdahl's law (3)



实际上并行加速不仅受限于串行分量，而且也受并行实现时的额外开销的限制。

John Gustafson (1955—)



Dr. Gustafson is an American computer scientist and businessman, chiefly known for his work in High Performance Computing (HPC) such as the invention of Gustafson's Law, introducing the first commercial computer cluster, etc.

Gustafson定律 (1)

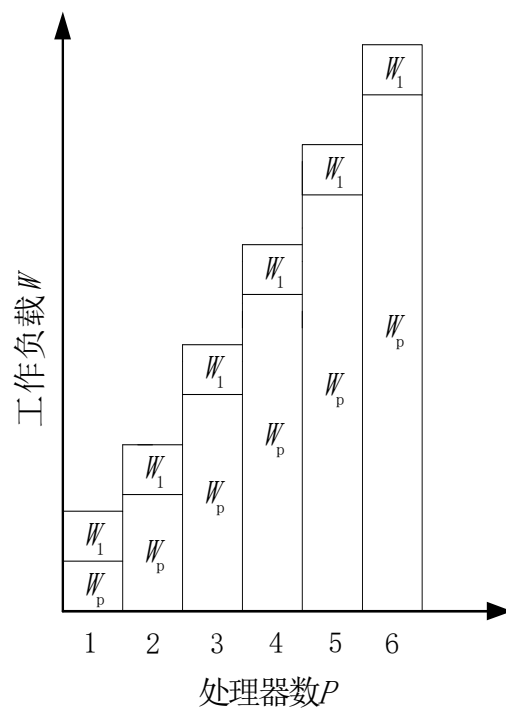
- 出发点：Base on Fixed Execution Time
 - 对于很多大型计算，精度要求很高，即在此类应用中精度是个关键因素，而计算时间是固定不变的。此时为了提高精度，必须加大计算量，相应地亦必须增多处理器数才能维持时间不变；
 - 表明：随着处理器数目的增加，串行执行部分 f 不再是并行算法的瓶颈。

- Gustafson加速定律：
$$S' = \frac{W_S + pW_P}{W_S + p \cdot W_P / p} = \frac{W_S + pW_P}{W_S + W_P}$$

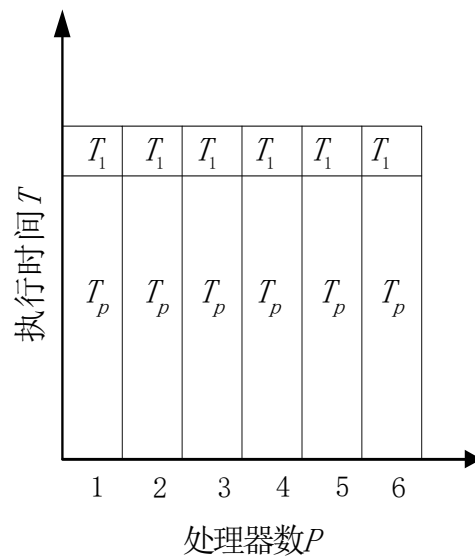
$$S' = f + p(1-f) = p + f(1-p) = p - f(p-1)$$

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

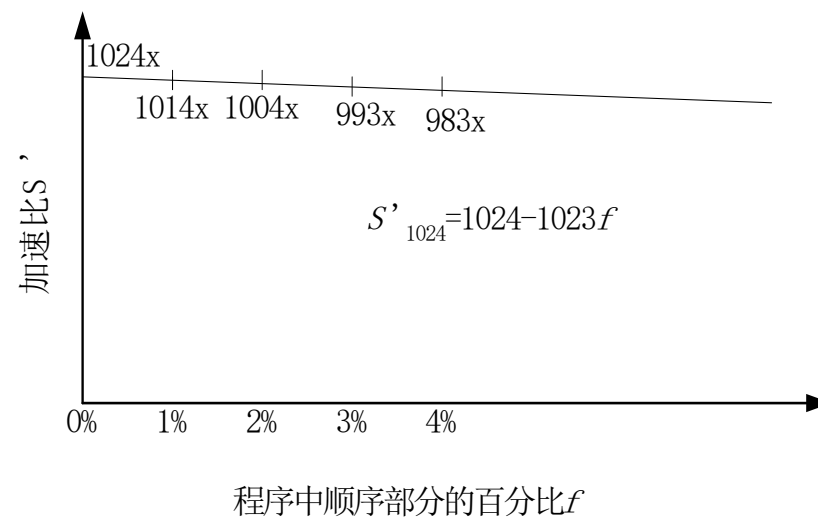
Gustafson定律 (2)



(a)



(b)



(c)

孙贤和 (Xian-He Sun)、倪明选 (Lionel M. Ni)



Sun 和 Ni定律 (1)

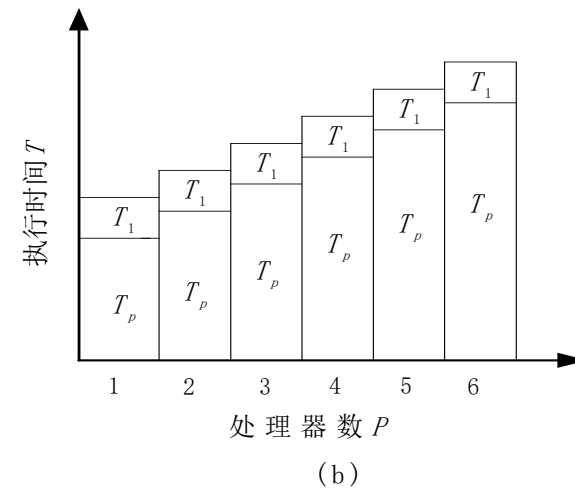
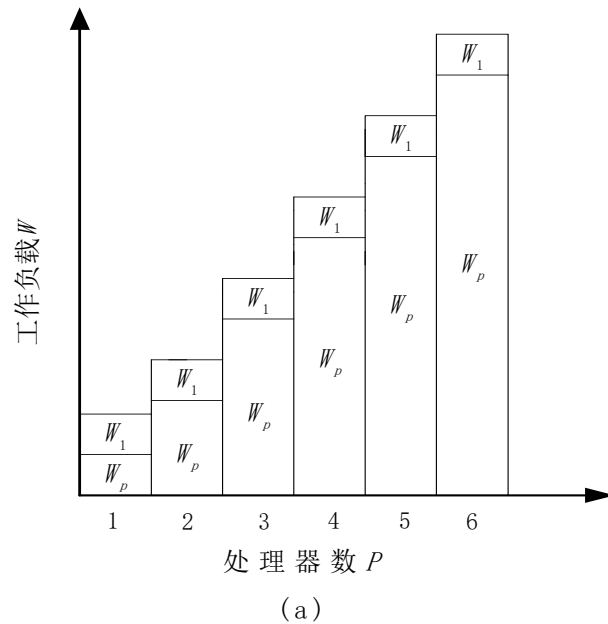
- 出发点: Base on Memory Bounding
 - 充分利用存储空间等计算资源, 尽量增大问题规模以产生更好/更精确的解。是Amdahl定律和Gustafson定律的推广。
- 公式推导:
 - 设单机上的存储器容量为M, 其工作负载 $W=fW+(1-f)W$
当并行系统有 p 个结点时, 存储容量扩大了 pM , 用 $G(p)$ 表示系统的存储容量增加 p 倍时工作负载的增加量。则存储容量扩大后的工作负载为 $W=fW+(1-f)G(p)W$, 所以存储受限的加速为

$$S'' = \frac{fW + (1-f)G(p)W}{fW + (1-f)G(p)W / p} = \frac{f + (1-f)G(p)}{f + (1-f)G(p) / p}$$

- 并行开销 W_o :

$$S'' = \frac{fW + (1-f)WG(p)}{fW + (1-f)G(p)W / p + W_o} = \frac{f + (1-f)G(p)}{f + (1-f)G(p) / p + W_o / W}$$

Sun 和 Ni 定律 (2)



- $G(p) = 1$ 时就是 Amdahl 加速定律;
- $G(p) = p$ 变为 $f + p(1-f)$, 就是 Gustafson 加速定律
- $G(p) > p$ 时, 相应于计算机负载比存储要求增加得快, 此时 Sun 和 Ni 加速均比 Amdahl 加速和 Gustafson 加速为高。

加速比讨论

- 参考的加速经验公式： $p/\log p \leq S \leq P$
 - 线性加速比：很少通信开销的矩阵相加、内积运算等
 - $p/\log p$ 的加速比：分治类的应用问题
- 通信密集类的应用问题： $S = 1 / C(p)$
这里 $C(p)$ 是 p 个处理器的某一通信函数
- 超线性加速
- 绝对加速：最佳串行算法与并行算法
- 相对加速：同一算法在单机和并行机的运行时间

第四章 并行计算性能评测

- 4.1 并行机的一些基本性能指标
- 4.2 加速比性能定律
 - 4.2.1 Amdahl定律
 - 4.2.2 Gustafson定律
 - 4.2.3 Sun和Ni定律
- 4.3 可扩展性评测标准
 - 4.3.1 并行计算的可扩展性
 - 4.3.2 等效率度量标准
 - 4.3.3 等速度度量标准
 - 4.3.4 平均延迟度量标准
- 4.4 基准测试程序

可扩放性评测标准（1）

- 并行计算的可扩放性（Scalability）也是主要性能指标
 - 可扩放性最简朴的含意是在确定的应用背景下，计算机系统（或算法或程序等）性能随处理器数的增加而按比例提高的能力
- 影响因素：处理器数与问题规模，还有
 - 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）；
 - 加大的处理器数超过了算法中的并发程度；
- 增加问题规模的好处：
 - 提供较高的并发机会；
 - 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）。
- 增加处理器数会增大额外开销和降低处理器利用率，所以对于一个特定的并行系统（算法或程序），它们能否有效利用不断增加的处理器能力应是受限的，而度量这种能力就是可扩放性这一指标。

可扩放性评测标准（2）

- 可扩放性:调整什么和按什么比例调整
 - 并行计算要调整的是处理数 p 和问题规模 W ,
 - 两者可按不同比例进行调整, 此比例关系 (可能是线性的, 多项式的或指数的等) 就反映了可扩放的程度。
- 与并行算法和体系结构相关
- 可扩放性研究的主要目的:
 - 确定解决某类问题用何种并行算法与何种并行体系结构的组合, 可以有效地利用大量的处理器;
 - 对于运行于某种体系结构的并行机上的某种算法当移植到大规模处理机上后运行的性能;
 - 对固定的问题规模, 确定在某类并行机上最优的处理器数与可获得的最大的加速比;
 - 用于指导改进并行算法和并行机体系结构, 以使并行算法尽可能地充分利用可扩充的大量处理器
- 目前无一个公认的、标准的和被普遍接受的严格定义和评判它的标准

等效率度量标准 (1)

- 令 t_{ie} 和 t_{io} 分别是并行系统上第 i 个处理器的有用计算时间和额外开销时间（包括通信、同步和空闲等待时间等）

$$T_e = \sum_{i=0}^{p-1} t_e^i = T_s \quad T_o = \sum_{i=0}^{p-1} t_o^i$$

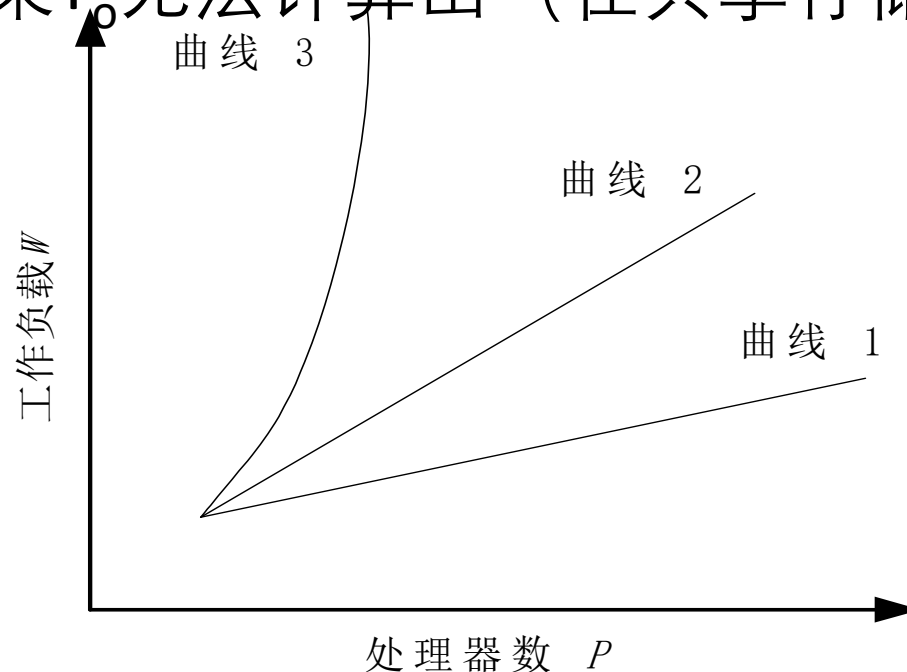
- T_p 是 p 个处理器系统上并行算法的运行时间，对于任意 i 显然有 $T_p = t_{ie} + t_{io}$ ，且 $T_e + T_o = pT_p$
- 问题的规模 W 定义为最佳串行算法所完成的计算量， $W = T_e$

$$S = \frac{T_e}{T_p} = \frac{T_e}{\frac{T_e + T_o}{p}} = \frac{p}{1 + \frac{T_o}{T_e}} = \frac{p}{1 + \frac{T_o}{W}} \quad E = \frac{S}{p} = \frac{1}{1 + \frac{T_o}{T_e}} = \frac{1}{1 + \frac{T_o}{W}}$$

- 如果问题规模 W 保持不变，处理器数 p 增加，开销 T_o 增大，效率 E 下降。为了维持一定的效率（介于0与1之间），当处理数 p 增大时，需要相应地增大问题规模 W 的值。由此定义函数 $f_E(p)$ 为问题规模 W 随处理器数 p 变化的函数，为等效率函数（ISO-efficiency Function）（Kumar1987）

等效率度量标准 (2)

- 曲线1表示算法具有很好的扩放性；曲线2表示算法是可扩放的；曲线3表示算法是不可扩放的。
- 优点：简单可定量计算的、少量的参数计算等效率函数
- 缺点：如果 T_0 无法计算出（在共享存储并行机中）



等速度度量标准 (1)

- 出发点：对于共享存储的并行机， T_0 难以计算，如果速度能以处理器数的增加而线性增加，则说明系统具有很好的扩放性。
- p 表示处理器个数， W 表示要求解问题的工作量或称问题规模（在此可指浮点操作个数）， T 为并行执行时间，定义并行计算的速度 V 为工作量 W 除以并行时间 T
- p 个处理器的并行系统的平均速度定义为并行速度 V 除以处理器个数 p ：

$$\bar{V} = \frac{V}{p} = \frac{W}{pT}$$

- W 是使用 p 个处理器时算法的工作量，令 W' 表示当处理器数从 p 增大到 p' 时，为了保持整个系统的平均速度不变所需执行的工作量，则可得到处理器数从 p 到 p' 时平均速度可扩放度量标准公式（介于0与1之间，比值越靠近1越好）

$$\Psi(p, p') = \frac{W/p}{W'/p'} = \frac{p'W}{pW'}$$

等速度度量标准（2）

- 优点：直观地使用易测量的机器性能速度指标来度量
- 缺点：某些非浮点运算可能造成性能的变化没有考虑
- 等速度度量标准的扩放性与传统加速比之间的关系：
当 $p=1$ 时，等速度度量标准为

$$\Psi(p') = \Psi(1, p') = \frac{W/1}{W'/p'} = \frac{p'W}{W'} = \frac{T_1}{T_{p'}}$$

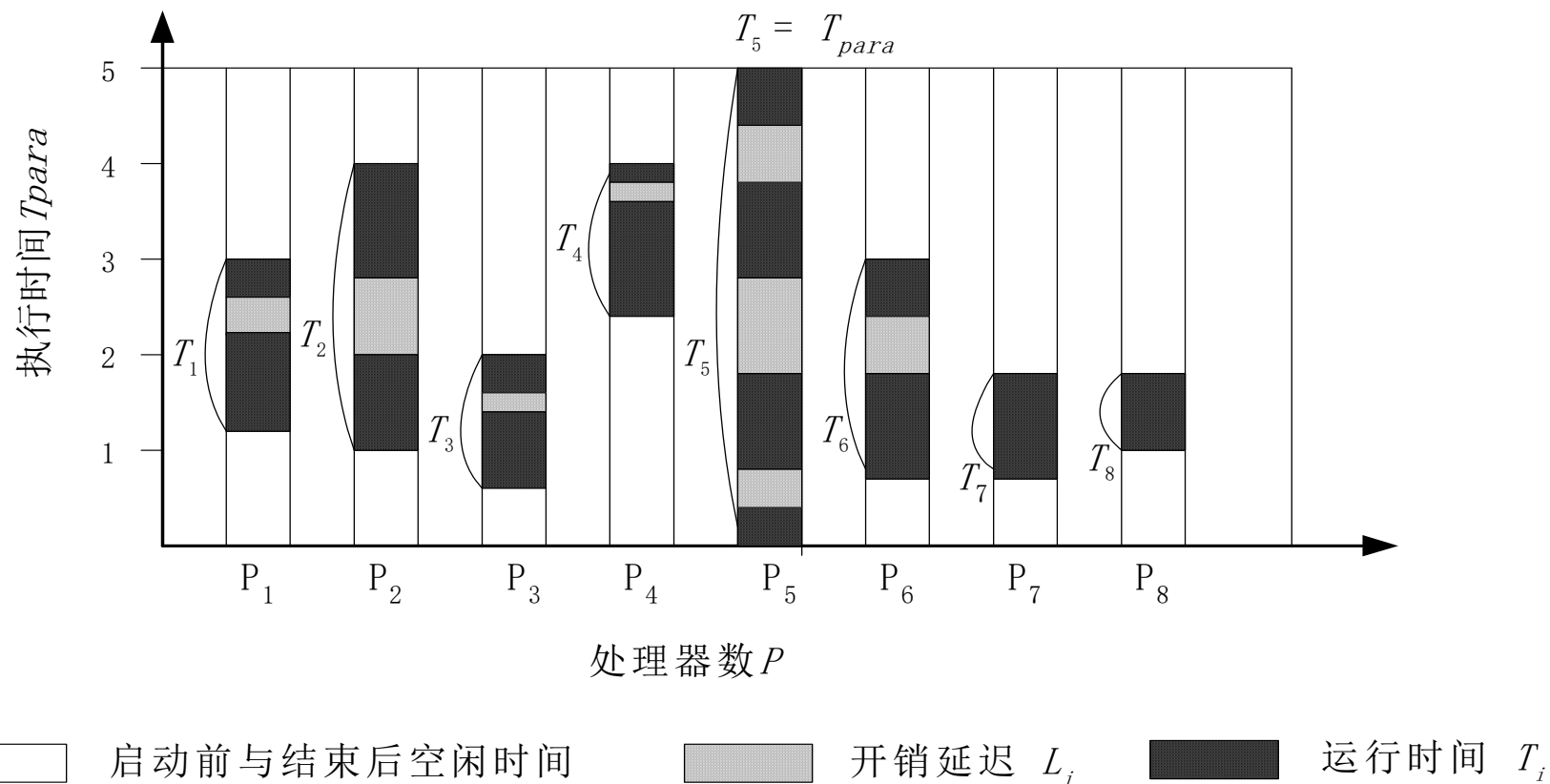
= $\frac{\text{解决工作量为}W\text{的问题所需串行时间}}{\text{解决工作量为}W'\text{的问题所需并行时间}}$

其主要差别：

加速比的定义是保持问题规模不变，标志对于串行系统的性能增加；扩放性定义是保持平均速度不变，标志对于小系统到大规模系统所引起的性能变化

平均延迟度量标准 (1)

- 一个并行系统执行的时间图谱



平均延迟度量标准 (2)

- T_i 为 P_i 的执行时间, 包括延迟 L_i 。 P_i 的总延迟时间为“ L_i + 启动时间 + 停止时间”。 定义系统平均延迟时间为

$$\bar{L}(W, p) = \sum_{i=1}^p (T_{para} - T_i + L_i) / p$$

$$\text{又有 } pT_{para} = T_o + T_{seq}^1, \quad T_o = p\bar{L}(W, p)$$

$$\text{所以 } \bar{L}(W, p) = T_{para} - T_{seq} / p$$

- 在 p' 个处理器上求解工作量为 W' 问题的平均延迟 $\bar{L}(W', p')$
- 当处理器数由 p 变到 p' , 而维持并行执行效率不变, 则定义平均延迟可扩放性度量标准为

$$\Phi(E, p, p') = \frac{\bar{L}(W, p)}{\bar{L}(W', p')}$$

该值在 0、1 之间, 值越接近 1 越好。

平均延迟度量标准（3）

- 优点：平均延迟能在更低层次上衡量机器的性能
- 缺点：需要特定的软硬件才能获得平均延迟

第四章 并行计算性能评测

- 4.1 并行机的一些基本性能指标
- 4.2 加速比性能定律
 - 4.2.1 Amdahl定律
 - 4.2.2 Gustafson定律
 - 4.2.3 Sun和Ni定律
- 4.3 可扩展性评测标准
 - 4.3.1 并行计算的可扩展性
 - 4.3.2 等效率度量标准
 - 4.3.3 等速度度量标准
 - 4.3.4 平均延迟度量标准
- 4.4 基准测试程序

Performance profiling, modeling and auto-tuning for parallel programs

孙经纬

算法与数据应用研究组 中国科学技术大学

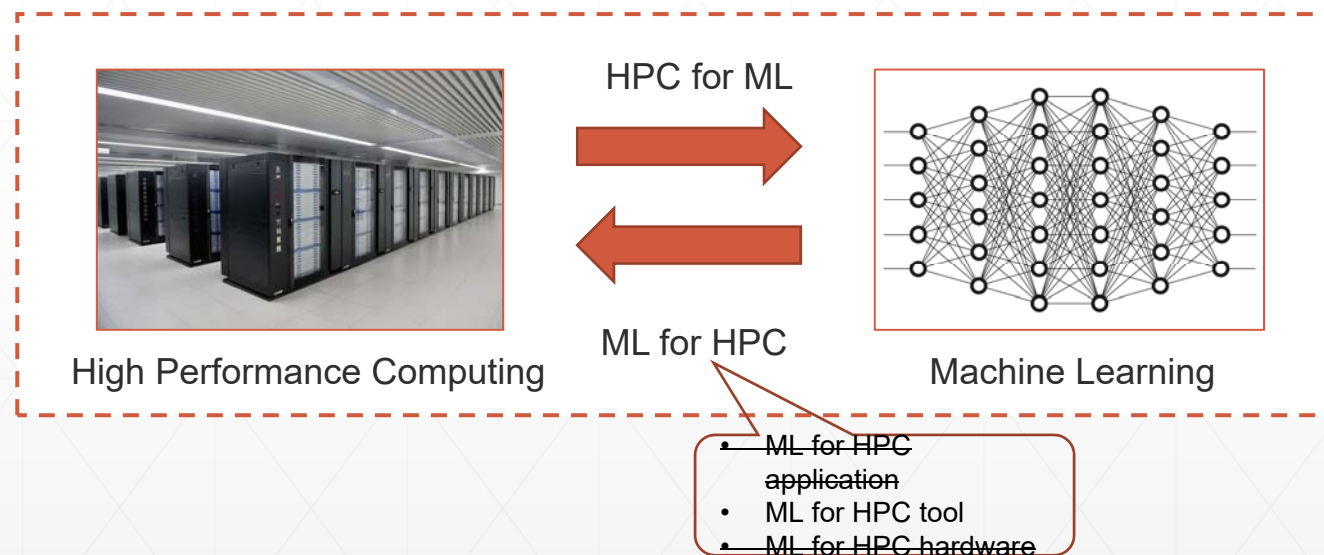
2022-02-28

内容提纲

- 研究方向
- 研究内容
 - Automatic Synthesis of MPI Benchmarks
- 后续计划


研究方向

- High performance computing, parallel computing
- Performance profiling, modeling, auto-tuning



性能评测

- 高性能计算领域中最基础的问题之一
- 设计和优化各种体系结构、系统软件以及高性能算法的前提



HOME LISTS STATISTICS RESOURCES ABOUT MEDIA KIT

Home » Lists » Top500 » November 2021 » List

TOP500 LIST - NOVEMBER 2021

R_{max} and **R_{peak}** values are in TFlops. For more details about other fields, check the TOP500 description.

R_{peak} values are calculated using the advertised clock rate of the CPU. For the efficiency of the systems you should take into account the Turbo CPU clock rate where it applies.

← 1-100 101-200 201-300 301-400 401-500 →

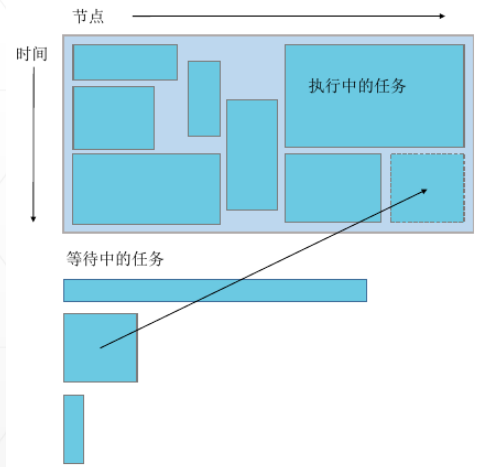
Rank	System	Cores	R _{max} (TFlop/s)	R _{peak} (TFlop/s)	Power (kW)
1	Supercomputer Fugaku - Supercomputer Fugaku, A64FX 48C 2.26GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory	2,414,592	148,600.0	200,794.9	10,096
10	Voyager-EUS2 - ND96amsr_A100_v4, AMD EPYC 7V12 48C 2.45GHz, NVIDIA A100 80GB, Mellanox HDR Infiniband, Microsoft Azure Azure East US 2 United States	253,440	30,050.0	39,531.2	

LINPAC
K跑分性能

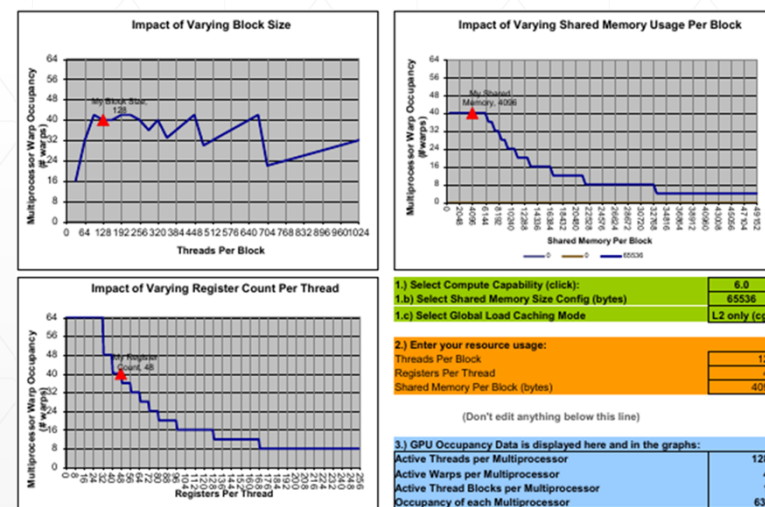
性能建模

- 基于评测与分析的结果，构建性能模型，回答这些问题：
- 程序需要多少执行时间（存储、带宽、能耗.....）？
- 哪些因素影响了执行时间？怎样影响？

根据任务的预测执行时间，
可以使用backfiling提高系
统利用率



Block size, shared memory/block,
register/thread 对GPU占用率的影响



性能模型分类

- 解析模型 (Analytical Model)

根据HPC程序的领域知识, 建立性能描述公式

算法的复杂度分析

Case by case

- 统计模型 (Statistical Model)

收集HPC程序的历史执行数据

拟合特征与性能间的相关性, 建立预测模型

- 回放模型 (Replay-based Model)

执行benchmark (proxy, mini-app), 获得程序的性能指标

研究重点

内容提纲

- 研究方向
- 研究内容
 - Automatic Synthesis of MPI Benchmarks
- 后续计划

Benchmark

- 基准测试程序 (Benchmark) :

用于评测特定Workload在给定平台 (硬件+运行时环境) 上的性能的软件

- 例1. LINPACK —— 计算密集型的线性代数程序
- 例2. Graph500 —— 数据密集型的图处理程序

- Benchmark相比于直接测量原应用程序的优点:

- 作为具有代表性的简化性能代理 (performance proxy) , 时间和硬件资源开销通常更低
- 避免复杂的移植 (安装依赖库、配置运行环境、准备数据集等)
- 避免商业软件、保密软件的授权费用、代码闭源、数据涉密等问题

Benchmark开发的困难

1. 人工开发

- 需要同时懂高性能计算和领域应用（物理、化学、生物、人工智能.....）的专家

2. 滞后于应用发展

- 应用本身的重要性更高，分配到Benchmark开发上的人力物力有限

3. 准确性不足

- 应用和系统都十分复杂，两者之间的交互更加复杂
- 专家开发的Benchmark也不一定能准确、全面地反映应用在系统上的性能特征

AlexNet: 2012	MLPerf: 2018
ResNet: 2015	MLBench: 2018
Transformer: 2017	AlBench: 2018
	SuperBench(v0.4): 2021

Benchmark自动生成 —— 用数据和程序构造Benchmark

- 主要针对CPU集群上的MPI 程序开展
 - MPI是目前主流的HPC编程标准
- 总体思路
 - 获取MPI程序运行时事件的参数和性能信息 (trace)
 - 将trace转换为可执行代码或可解释命令序列

The diagram shows a sample of MPI communication trace data. It includes a box for '进程id' (Process ID) pointing to 'Rank=0', a box for '函数名 (事件类型)' (Function Name / Event Type) pointing to 'Function: MPI_Isend', a box for '参数信息' (Parameter Information) pointing to the 'Paravalues' block, and a box for '时间戳' (Timestamp) pointing to the 'Starttime' and 'Endtime' fields.

```
Rank=0 Function: MPI_Isend
Paravalues: = (
    mpip_const_void_t buf=-1270915064,
    int count= 961,
    MPI_Datatype datatype=1275070475,
    int dest= 4,
    int tag= 1024,
    MPI_Comm comm=1140850688,
    MPI_Request request=160750800 )
Starttime = [ 1554884741014773 ]
Endtime = [ 1554884741014789 ]
```

MPI通信trace示例

关键问题

1. Trace的平台无关 (Platform-independent) 表达

- 通信trace:

- MPI本身是由平台无关的通信原语组成, 无需特别处理

- 计算trace:

- 单个计算操作粒度过细, 跟踪记录开销大, 实际上也无法准确记录 (编译优化、流水线、乱序执行.....)
 - Profiling可以获得一段代码的计算性能指标, 但这是计算操作的结果, 不是计算操作本身, 而且平台依赖
 - 抽象为这样一个问题: 给定一组性能指标, 求一段符合这些指标的**高级语言代码的参数化表达**

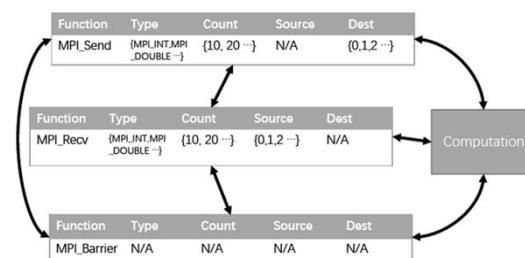
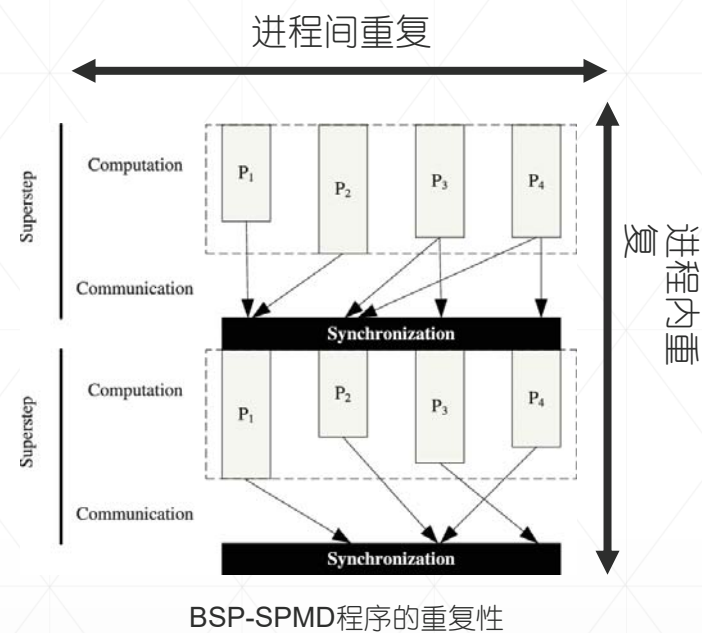
平台无关

便于表格存储
以及作为特征

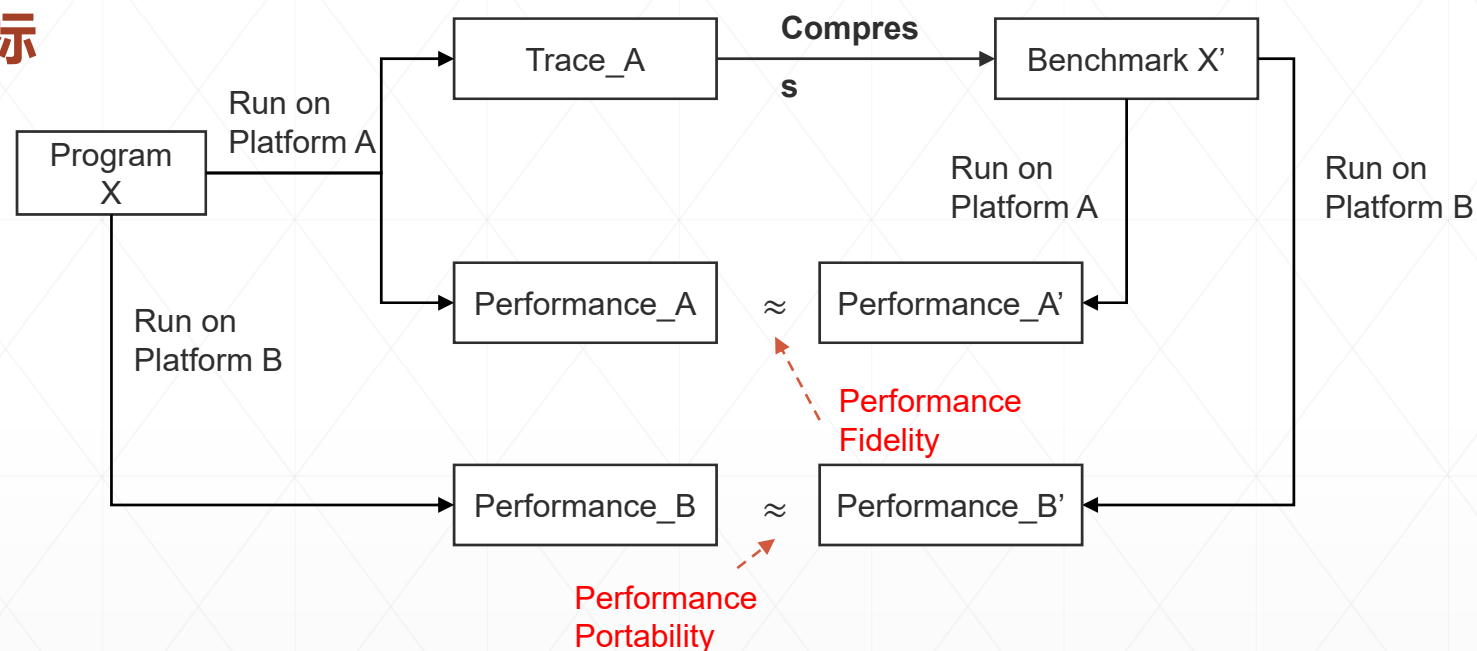
关键问题

2. Trace压缩

- Trace需要大量的存储空间
 - 例如，简单的SMG2000程序在运行中可以产生5TB 通信trace数据
- 结构化压缩
 - 针对BSP-SPMD程序（ Bulk Synchronous Parallel, Single Program Multiple Data ）
- 非结构化压缩
 - 跟结构化压缩相反
 - 认为程序运行是在不同事件之间随机游走， trace是采样的结果



预期目标



- Performance Fidelity: 基于平台A上的Trace_A生成的Benchmark，在平台A上执行得到的性能指标和原程序相似
- Performance Portability: 相似性在相同并行规模的平台B上仍然保持
- ~~Performance Scalability: 相似性在不同并行规模下仍然保持~~ 目前还做不到，也没想到有什么办法可以做到

技术路线

1. Trace生成

- 通信trace
- 计算trace

2. Trace分类

- 定量描述并行程序的行为模式

3. Trace压缩

- 结构化压缩
- 非结构化压缩

4. Trace回放

通信trace

- 利用MPI性能分析层（PMPI）追踪和记录通信事件
- 为每个MPI函数创建wrapper
- 添加MPI函数执行前后的信息记录和分析操作

```
/* ----- MPI_Send ----- */  
  
static int mpiPif_MPI_Send( jmp_buf * base_jbuf, mpip_const_void_t *  
{  
    int DT_size=0;  
    double start_from_begin, end_from_begin, overhead; char event_buffer;  
    int rc, enabledState;  
    double dur;  
    int tsize;  
    double messSize = 0.;  
    double ioSize = 0.;  
    double rmaSize = 0.;  
    mpiPi_TIME start, end, end2;  
    void *call_stack[MPIP_CALLSITE_STACK_DEPTH_MAX] = { NULL };  
    mpiPi_mt_stat_tls_t *hdl;  
  
    hdl = mpiPi_stats_mt_gettls(&mpiPi.task_stats);  
  
    if (mpiPi_stats_mt_is_on(hdl)) {  
        mpiPi_GETTIME (&start);  
        if ( mpiPi.reportStackDepth > 0 ) mpiPi_RecordTraceBack((*base_jbuf),  
    }  
  
    mpiPi_stats_mt_enter(hdl);  
  
    rc = PMPI_Send( buf, * count, * datatype, * dest, * tag, * comm;  
  
    mpiPi_stats_mt_exit(hdl);  
    if (mpiPi_stats_mt_is_on(hdl)) {  
  
        mpiPi_GETTIME (&end);  
        dur = mpiPi_GETTIMEDIFF (&end, &start);  
        start_from_begin = mpiPi_GETTIMEDIFF (&start, &mpiPi.time_begin);  
        end_from_begin = mpiPi_GETTIMEDIFF (&end, &mpiPi.time_begin);  
        MPI_Type_size(*datatype, &DT_size);  
        sprintf(event_buffer, "%d,MPI_Send,%d;%d;%d,%.01f,%.01f\n", mpiPi.rank,  
        list_push(&mpiPi.event_list, event_buffer);
```

计算trace

- PAPI性能分析库
 - 对程序某一段程序的使用时钟周期数，执行指令数，L1/L2 cache miss/access数，TLB miss数等进行统计，提供直观的程序的局部计算性能信息
 - 每个进程维护自身的一组计数器，彼此之前互不影响
 - 在上一个通信事件结束后开始硬件计数，到下一个通信事件开始结束计数
- 原生事件（Native Event）：
 - 不同的处理器会根据自身的体系结构特征定义出不同的处理器事件集合
- 预设事件（Preset Event）：
 - PAPI 将不同处理器中存在功能共性的原生事件统一接口，实现跨平台的效果

计算trace

- 给定一组性能指标，求一段符合这些指标的高级语言代码的参数化表达（还在寻找更好的办法）
 - 假设待求的C语言代码由 n 个for循环（无嵌套）拼接组成，不考虑循环之间的顺序
 - 每个循环体内都是手工预设的代码，对性能指标 p_1, p_2, \dots, p_m 有不同程度升高或降低效果
 - 通过调整第 i 个循环的循环次数 x_i ，调整该循环对某些性能指标的影响程度
 - 求一组最优的循环次数，使得该段C代码编译执行后的性能指标与Compute参数的距离最小

```
1 int i;
2 int a = 3, b = 21, c;
3 for(i = 0; i < iternum; i++){
4     /* code for increment ipc */
5     c = a + b;
6     c = a + b;
7     .....
8     c = a + b;
9     c = a + b;
10 }
```

图 3.1 增加 IPC 的代码块

```
1 int i;
2 double da = 222, db = 3, dc;
3 for(i = 0; i < iternum; i++){
4     /* code for decrement ipc */
5     dc = da / db;
6     dc = da / db;
7     .....
8     dc = da / db;
9     dc = da / db;
10 }
```

图 3.2 减小 IPC 的代码块

```
1 int i;
2 int *array_inc, N = (1024 * 1024) * 4;
3 array_inc = (int *)malloc(N * sizeof(int));
4 for(i = 0; i < 10000; i++){
5     /* code for increment cmr */
6     array_inc[rand() % N] = 0;
7 }
8 }
```

图 3.3 增加 CMR 的代码块

```
1 int i;
2 int N = 256, j, k;
3 int array_dec[N][N];
4 for(i = 0; i < iternum; i++){
5     /* code for decrement cmr */
6     for(j = 0; j < N; j++){
7         for(k = 0; k < N; k++){
8             array_dec[j][k] = array_dec[j][k] * 2;
9         }
10     }
11 }
```

图 3.4 减小 CMR 的代码块

```
1 int i;
2 int a2 = 222, b2 = 3, c2;
3 int randnum, r3, r4, r8;
4 for(i = 0; i < iternum; i++){
5     /* code for increment bmr */
6     randnum = rand();
7     r3 = randnum % 3;
8     if(r3 == 0) c2 = b2 + a2 + (b2 / a2);
9     if(r3 == 1) c2 = b2 + a2 + (b2 / a2);
10    r4 = randnum % 4;
11    if(r4 == 0) c2 = b2 + a2 + (b2 / a2);
12    if(r4 == 1) c2 = b2 + a2 + (b2 / a2);
13    r8 = randnum % 8;
14    if(r8 == 0) c2 = b2 + a2 + (b2 / a2);
15    if(r8 == 1) c2 = b2 + a2 + (b2 / a2);
16    if(r8 == 2) c2 = b2 + a2 + (b2 / a2);
17    if(r8 == 3) c2 = b2 + a2 + (b2 / a2);
18 }
```

图 3.5 增加 BMR 的代码块

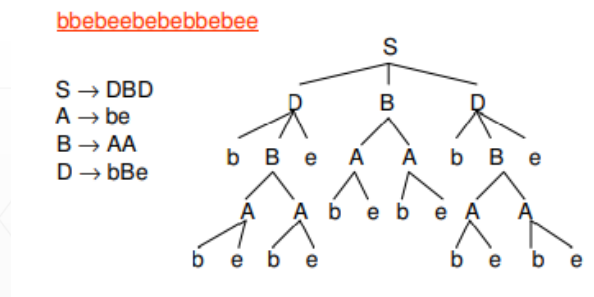
```
1 int i;
2 int a1 = 222, b1 = 3, c1;
3 for(i = 0; i < iternum; i++){
4     /* code for decrement bmr */
5     c1 = b1 + a1 + (b1 / a1);
6     c1 = b1 + a1 + (b1 / a1);
7 }
8 }
```

图 3.6 减小 BMR 的代码块

Instructions Per Cycle, Cache Miss Rate,
Branch Misprediction Rate预设代码示例

并行程序行为模式分类

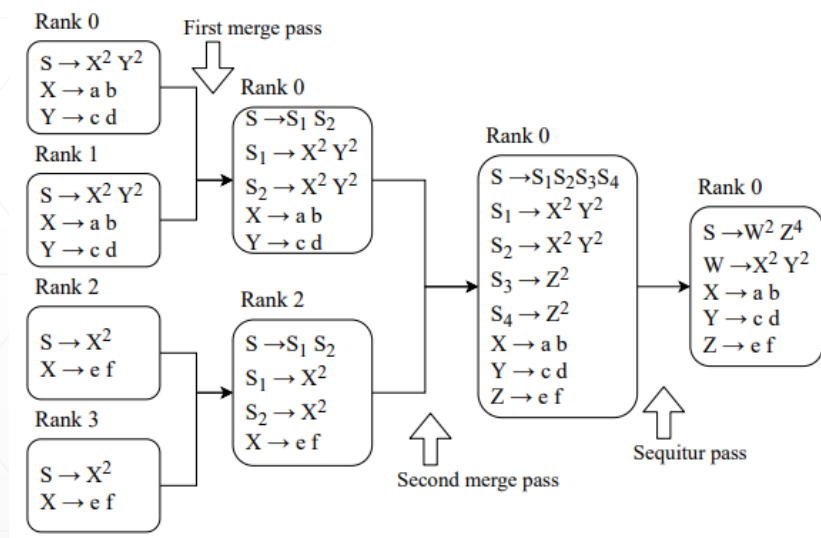
- 将每个事件看成一个符号，trace是一个符号序列（字符串）
 - 符号表记录每个符号的含义，如函数名、类型、参数取值等（字母表）
 - Context Free Grammar (CFG) 表达符号序列的循环和嵌套结构



- On-the-fly模式：MPI程序执行的同时产生trace和更新CFG产生式

并行程序行为模式分类

- 根据产生式数量可以判断行为的结构化程度
 - 进程内产生式越少，说明具有稳定的重复迭代计算行为
 - 进程间合并后产生式越少，说明越符合BSP-SPMD
- 设定产生式数量阈值
 - 大于阈值，停止继续构造CFG，采用非结构化压缩
 - 小于阈值，采用结构化压缩



Trace压缩

- 结构化压缩
 - CFG构造完成时，已经完成了结构化压缩
 - CFG很容易转换成程序代码
- 非结构压缩：不假定程序行为模式，不显式搜索和构造重复结构
 - MPI 事件 → 单词
 - 事件序列 → 语句
 - Trace → 语料库
 - 压缩trace → 训练一个语言模型
 - 回放 → 文本生成

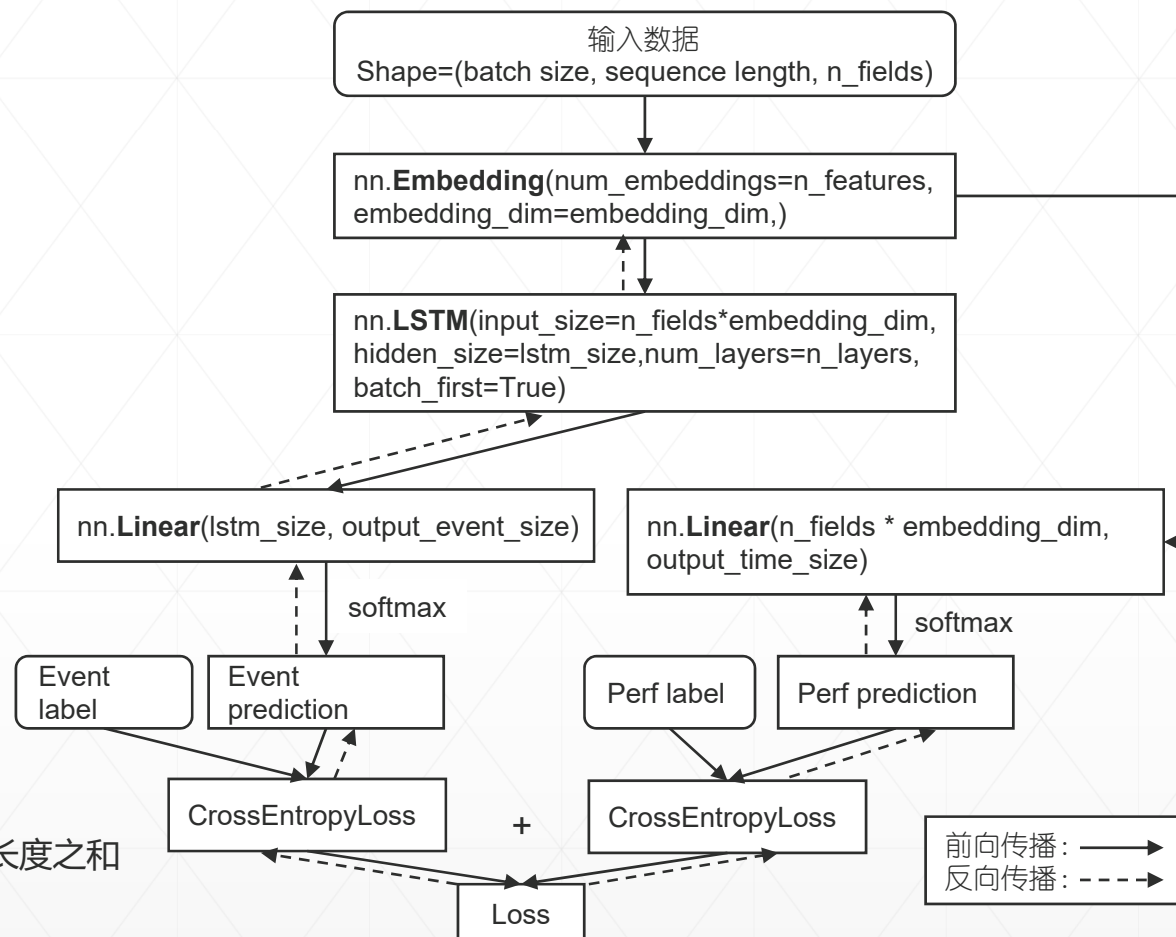
Trace非结构压缩

- 模型包含两个预测任务组：

- MPI事件序列
- 事件的性能指标

- n_fields : 事件特征数量

- $n_features$: one-hot事件特征长度之和



实验结果

测试程序

- HPC科学计算kernel和mini-app: NPB (BT、CG、MG、SP) 、LULESH、SWEEP3D

Baselines

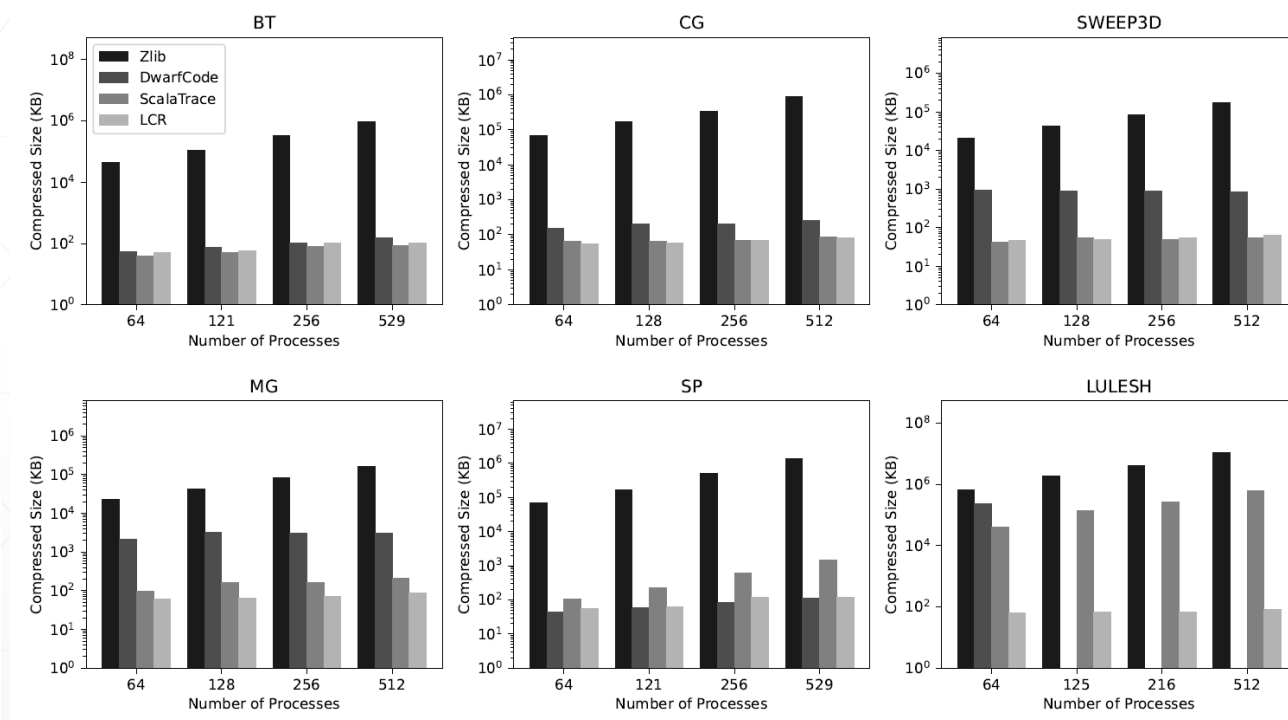
- Zlib: 通用的数据压缩库, 被Open Trace Format采用, 具体的算法是LZ77和Huffman coding结合。
- DwarfCode: 针对MPI trace的无损压缩算法, 将trace压缩问题转换为一个NP-hard的组合优化问题, 并采用一种近似算法求解次优解。
- ScalaTrace: 针对MPI trace的有损压缩算法, 在结构化匹配trace中的相似模式的同时, 将MPI函数参数以及时间间隔用histogram代替具体数值。

实验结果

Trace specification

Program	# Process	# Events	Size (MB)
BT	64	2,909,312	1,032
	121	7,687,130	2,713
	256	23,974,400	8,414
	529	71,847,722	25,069
CG	64	5,100,288	1,815
	128	13,280,896	4,715
	256	26,529,024	9,435
	512	65,313,792	23,298
MG	64	1,641,504	588
	128	3,101,856	1,111
	256	6,032,352	2,164
	512	11,912,928	4,264
SP	64	4,458,496	1,783
	121	11,702,878	4,673
	256	36,302,848	14,456
	529	108,410,086	42,945
SWEEP3D	64	1,795,520	713
	128	3,719,040	1,477
	256	7,694,080	3,062
	512	15,644,160	6,225
LULESH	64	38,649,272	14,395
	125	106,005,722	39,587
	216	215,652,072	80,725
	512	557,271,432	209,012

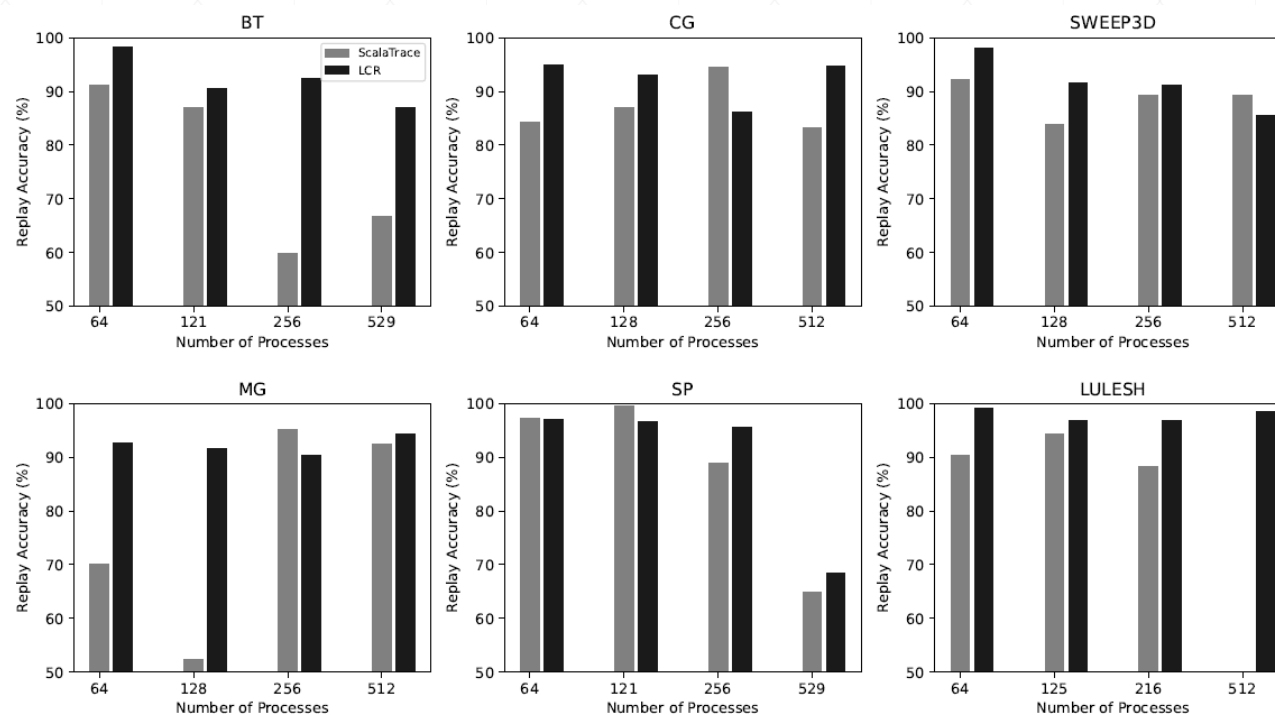
将500MB~200GB的trace转换为不到100KB的模型



实验结果

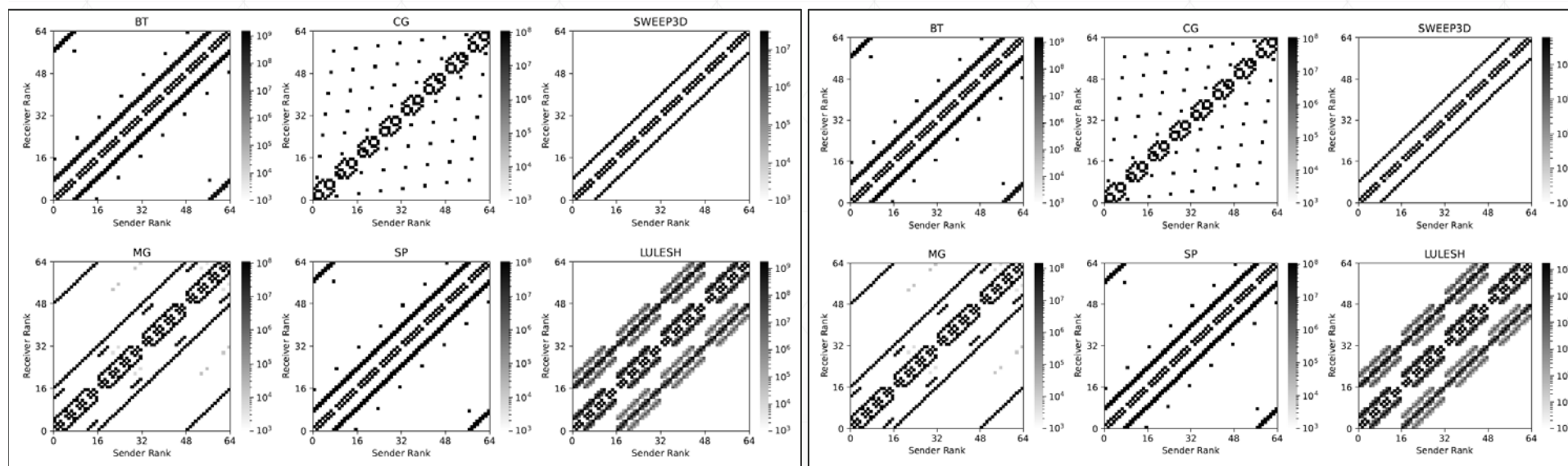
回放效果

- 与SOTA方法ScalaTrace对比回放时间准确度
- 总体准确度更高



实验结果

- 回放效果：进程间p2p通信数据量可视化



实际的通信数据量

回放的通信数据量

内容提纲

- 研究方向
- 研究内容
 - Automatic Synthesis of MPI Benchmarks
- 后续计划

作业

- 139页： 4.2 中 (1) 和 (2)
- 140页： 4.11
- 141页： 4.14
- BB系统提交
- 截止时间2022年3月18日(周五)18:00