

Tradeoff Speed/Power: por causa da energia gasta e do calor dissipado, não vale a pena aumentar a frequência do processador

• O desafio principal é escrever programas escaláveis que mantêm o nível de eficiência conforme os dados aumentam e conforme há mais "cores" disponíveis — os compiladores não conseguem programar processadores multicore

Como avaliar desempenho? "speedup"; eficiência; escalabilidade

• A evolução dos sistemas de computação é altamente paralela e heterogênea

• A simulação (numérica) é o terceiro pilar da ciência — computação científica

Porquê computação paralela?

- resolver problemas maiores e com uma representação mais realista
- reduzir custos de desenvolvimento
- ter mais liberdade para explorar alternativas

MIPS: milhões de instruções por segundo

FLOPS: operações de vírgula flutuante por segundo

R_{peak}: pico de desempenho — velocidade do CPU

R_{max}: desempenho máximo para um dado algoritmo

N_{max}: tamanho do problema para alcançar R_{max}

• O desempenho do computador depende de vários fatores: I/O, memória, ...

• O desempenho relevante é o que resulta da execução real de um algoritmo

- O desempenho sustentado também depende do design do algoritmo
- Uma implementação compatível com uma arquitetura de computadores pode ter o mesmo desempenho (sustentado) para um intervalo mais amplo de dados de input

Lei de Amdahl: numa aplicação, há sempre uma parte que não pode ser paralelizada — mesmo que a parte paralela seja perfeitamente escalável, o desempenho é limitado pela parte sequencial

λ : parte sequencial do trabalho

$(1-\lambda)$: parte do trabalho que pode ser paralelizada

P : número de processadores

Speedup: ganho obtido com o programa paralelo = T_1 / T_p

$$T_p = \frac{(1-\lambda)}{P} + \lambda$$

$$\text{Speedup} \leq \frac{1}{\frac{1-\lambda}{P} + \lambda} = \frac{1}{T_p}$$

• Se $P \rightarrow \infty$, então Speedup $\rightarrow 1/\lambda$

Código Inerentemente Paralelável: parte do código que executa com Speedup = P se correr em P processadores

Código Inerentemente Sequencial: parte do código que não pode ser paralelizada, como I/O ou inicialização de variáveis

Conclusão: A Lei de Amdahl ...

1. permite ter uma expectativa realista, para um dado algoritmo, sobre o que se consegue obter com uma execução paralela
2. mostra que para alcançar "speedups" mais altos é necessário reduzir ou eliminar os blocos sequenciais do algoritmo
3. dá uma métrica de comparação para medir o paralelismo de vários algoritmos para o mesmo problema

A função de Speedup cresce até um dado número de processadores P e degrada a partir daí — isto porque a parte inherentemente sequencial aumenta conforme o número de processadores aumenta

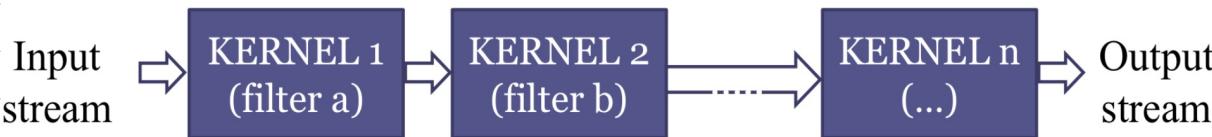
Paralelismo Funcional: Tarefas independentes executam operações diferentes em diferentes conjuntos de dados

Ex: $\text{sum} = 0$
 $\text{for } (i=0; i < n; i++) \text{ sum} += x[i]$

Paralelismo de Dados: Tarefas independentes executam a mesma operação sobre dados diferentes

Ex: $\text{for } (i=0; i < 99; i++) a[i] = b[i] + c[i]$

Streaming: processar fluxos de dados — dividir o processo num dado número de fases, que limita o Speedup (máximo)



$$\text{v}_{\text{SRAM}} > \text{v}_{\text{DRAM}} > \text{v}_{\text{DISCO}}$$

Memória Cache: interface entre o processador e a memória principal

Tempo de Execução: ciclos de relógio a executar código do utilizador + ciclos de relógio para transferência de dados entre cache e memória

Cache Hit: o CPU pede dados disponíveis na cache

Cache Miss: o CPU pede dados que não estão na cache

Localização Espacial: quando um elemento de dados é pedido, então os seus vizinhos também vão ser — uma linha de cache é lida numa única operação

Localização Temporal: quando um elemento de dados é pedido, então há alta probabilidade de voltar a ser pedido num curto período de tempo

Coerência de Cache: em processadores multicore, os cores partilham um espaço de endereçamento comum, mas as caches são independentes por core
→ **PROBLEMA:** dados partilhados

Sistema de Memória Coerente

1. Uma leitura depois de uma escrita numa localização X pelo mesmo processador P deve retornar a última escrita
2. Uma leitura por P depois de uma escrita por Q deve retornar a última escrita, se ambas as instruções não suficientemente separadas no tempo e nenhuma outra escrita ocorre por nenhum processador
3. Escritas para a mesma localização não serializadas — duas escritas por processadores diferentes não visitam na mesma ordem por todos os processadores

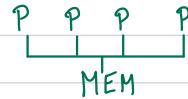
Protocolo "Write-Invalidade": para garantir coerência, garantir que um único processador tem acesso exclusivo à localização de memória em que quer escrever, invalidando todas as cópias que foram existentes noutros processadores, forçando-os a ler novamente o valor daquela região de memória

- Muitos protocolos usam blocos de dados em vez de localizações únicas de memória, mas isso pode levar a falsa partilha.

Modelos de Programação Partilhada

Modelo de Memória Partilhada: cada processador (ou core) executa uma thread — as threads interagem por variáveis partilhadas

- O número de forks/joins influencia o desempenho

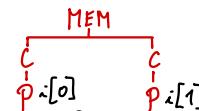


- Cada thread tem o seu próprio estado do processo ("program counter", "stack pointer" e variáveis locais), mas partilham variáveis globais definidas pela thread principal

OpenMP: ~~#pragma omp parallel for~~ NÚMERO DE THREADS
num-threads (k)
private (x) VARIÁVEL PRIVADA

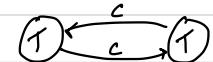
"Data Race": quando duas ou mais threads podem modificar a mesma localização de memória ao mesmo tempo

Seção Crítica: passão de código que só uma thread de cada vez pode executar
~~#pragma omp critical~~



Falsa Partilha: quando duas ou mais threads accedem a dados diferentes na mesma linha de cache para leitura ou escrita - o esforço necessário para manter a consistência de memória degrada o desempenho

Modelo de Memória Distribuída: modelo Tarefa/canal

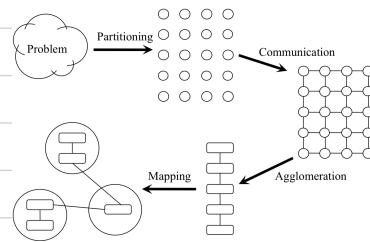


Programa Paralelo: conjunto de tarefas a executar concorrentemente

Tarefa: programa sequencial com memória local e um conjunto de portas I/O

As tarefas interagem através do envio de mensagens por canais de comunicação

1. Particionamento do Problema
2. Padrões de Comunicação
3. Aglomeração
4. Mapeamento



MapReduce: aplicar uma dada função para todos os elementos de dados e combinar esses resultados usando uma função definida pelo utilizador

1. Map: aplicar uma função definida pelo utilizador sobre um conjunto de pares chave-valor, gerando outro conjunto de pares (chave, valor) - paralelismo de dados

2. Reduce: aplicar uma função reduzora definida pelo utilizador a um conjunto de pares (chave, valor) onde os dados podem ser agregados, filtrados e combinados de várias formas - paralelismo funcional

Programação com Memória Partilhada

OpenMP: API para programação paralela em "multicore"

• Inicialmente, só a thread principal está ativa e executa código sequencial

"Fork": a thread principal cria ou acorda threads adicionais para executar código paralelo

"Join": no final do código paralelo, as threads criadas morrem ou são suspensas

• O número de threads ativas é 1 no início e no fim do programa, mas muda dinamicamente durante a execução

• Os programas paralelos com memória partilhada só podem ter um único ciclo paralelo

• O compilador gera código que faz "fork"/"join" das threads e aloca as iterações às threads

Pragma: diretiva para o programador comunicar com o compilador de C/C++
→ #pragma omp ...

Contexto de Execução: espaço de endereçamento que contém todas as variáveis que uma thread pode acessar — cada thread tem o seu contexto de execução

Variável Partilhada: tem o mesmo endereço no contexto de execução de cada thread

Variável Privada: tem um endereço diferente no contexto de execução de cada thread

• Uma thread não pode acessar às variáveis privadas de outra thread

`int omp_get_num_procs()`: retorna o número de processos físicos disponíveis para uso pelo programa paralelo

`void omp_set_num_threads(int t)`: define o número de threads a serem ativas em execuções paralelas do código

Bloco Paralelo: ~~#pragma~~ `omp parallel [num_threads(n)] { ... }`

Pragma "parallel": precede um bloco de código que deve ser executado por todas as threads - a execução é replicada entre todas as threads

• Deve-se paralelizar o ciclo externo para reduzir o número de "forks"/"joins"

Cláusula: componente adicional opcional de um pragma

Cláusula "private": instrui o compilador a fazer uma ou mais variáveis privadas
↳ private (lista de variáveis)

• As variáveis privadas são indefinidas na entrada da thread

Cláusula "firstprivate": cria variáveis privadas tendo valores iniciais idênticos à variável controlada pela thread principal quando se entra no ciclo

Cláusula "lastprivate": copia de volta para a thread principal a cópia privada da variável da thread que executou a instrução que ocorre em último quando o ciclo é executado sequencialmente

Pragma "critical": denota uma seção crítica (porção de código que só pode ser executada por uma thread de cada vez)

Cláusula "reduce": guarda resultados parciais em variáveis privadas e combina os resultados parciais depois do ciclo
reduction (of: var)

Cláusula "nowait": o compilador põe uma barreira de sincronização no final de cada instrução do ciclo paralelo

Blocos executados por uma única thread:

1. ~~pragma omp master~~
2. ~~pragma omp single~~
3. ~~pragma omp barrier~~

Demasiados "forks"/"joins" podem piorar o desempenho
COMO MELHORAR?

- Inverter ciclos se (1) o paralelismo for no ciclo interno e (2) depois da inversão, o ciclo externo pode ser feito paralelo
OU
- Definir a região paralela fora
OU
- Marcar as regiões paralelas

Cláusula "if": determina em tempo de execução se o ciclo deve ser executado em paralelo

Planeamento Estático: todas as iterações são alocadas a todos os threads, antes de qualquer iteração ser executada — baixo overhead; carga de trabalho desequilibrada

Planeamento Dinâmico: só algumas iterações são alocadas às threads no início da execução do ciclo, as iterações restantes são alocadas às threads que completam as iterações que lhes foram atribuídas — alto overhead; carga de trabalho equilibrada

Cláusula "schedule": especifica como é que as iterações de um ciclo devem ser alocadas às threads

schedule (type, [chunk])

o tamanho dos "chunks" vai diminuindo

Type: static; dynamic; guided; runtime

Chunk: intervalo contínuo de iterações

Ex:

1. (static): alocação de cerca de n/t iterações contíguas a cada thread
2. (static, C): alocação intercalada de "chunks" de tamanho C às threads
3. (dynamic): alocação dinâmica uma iteração de cada vez às threads
4. (dynamic, C): alocação dinâmica de C iterações de cada vez às threads
5. (guided): auto-escalonamento guiado com tamanho mínimo de chunk 1
6. (guided, C): auto-escalonamento guiado com tamanho mínimo de chunk C
7. (runtime): planeamento escolhido em tempo de execução (OMP_SCHEDULE)

int omp_get_thread_num(): retorna o ID da thread, ID $\in \{0, 1, \dots, t-1\}$

int omp_get_num_threads(): retorna o número de threads ativas

Pragma "parallel sections": precede um bloco de K blocos de código que podem ser executados concorrentemente por K threads

Pragma "section": precede cada bloco de código dentro do bloco precedido por ~~"~~ pragma `omp parallel sections`

Pragma "sections": aparece dentro de um bloco de código paralelo

O "pragma `omp section`" para a primeira seção paralela depois de "pragma `omp parallel sections`" pode ser omitido

Pragma "task": define uma tarefa que vai ser atribuída a uma thread da equipa de threads paralelos concorrentes

• Por defeito, as variáveis são firstprivate, pelo que é necessário partilhá-las explicitamente com a cláusula shared(vn)

• Não é necessário criar tarefas extra porque as tarefas-filho não executadas concorrentemente com o seu pai

Pragma "taskwait": especifica uma espera no completar de tarefas-filho geradas desde o início da tarefa atual

• Quando uma thread encontra uma diretiva "taskwait", a tarefa atual é suspensa até todos os tarefas-filho geradas antes da região "taskwait" completarem a sua execução

Pragma "taskgroup": quando uma thread encontra uma diretiva "taskgroup", começa a executar a região "taskgroup" e, no final dessa região "taskgroup", a tarefa atual é suspensa até todas as tarefas-filho geradas na região "taskgroup" e todos os seus descendentes completarem a sua execução

Pragma "taskloop": especifica que as iterações de um ou mais ciclos aninhados vão ser executados em paralelo usando tarefas explícitos — as iterações não distribuídas entre tarefas geradas pela diretiva e agendadas para serem executados

Nota: assim que uma região paralela é criada, nenhuma thread na equipa pode sair da região até ao fim da região e nenhuma thread pode juntar-se à região paralela

Sistemas Distribuídos

Sistema Distribuído: coleção de processos distintos que estão espacialmente separados e que comunicam uns com os outros através de troca de mensagens — um sistema é distribuído se o atraso de transmissão de mensagens não é negligenciável em comparação com o tempo entre eventos num único processo

Mensagem: sequência de bits cujo formato e significado não são especificados por um protocolo de comunicação e que é transportado da sua origem para o seu destino através de uma rede de comunicação.

Vantagens:

1. Partilha de Recursos
2. Acesso a Recursos Remotos
3. Desempenho
4. Escalabilidade
5. Tolerância a Falhas — Fielicidade e Disponibilidade

Desafios:

1. Centralização
2. Comunicação Síncrona
3. Segurança e (falta de) Confiança
4. Falhas Parciais
5. Latência IPC
6. Ausência de Tempo Global
7. Ausência de Memória Física Partilhada
8. Heterogeneidade

Canais de Comunicação

Assunções Erradas:

1. A rede é fiável
2. A latência é zero
3. A largura de banda é infinita
4. A rede é segura
5. A topologia não se altera
6. Há um administrador
7. O custo de transporte é zero
8. A rede é homogênea

PROPRIEDADES

"Connection-Based": os processos devem configurar o canal antes de trocarem dados
"Connectionless": os processos não têm que configurar o canal, podem trocar dados imediatamente

Fiável: garante que os dados enviados são entregues ao destino respetivo -
não, os processos de comunicação não são notificados

Não Fiável: os processos de comunicação devem detectar a perda de mensagens e proceder conforme requerido pela aplicação

Gera Duplicados: o canal pode entregar mensagens duplicadas ao destino - o receptor é que deve detectar os duplicados

Sem Duplicados: o canal garante que entrega cada mensagem aos seus receptores no máximo uma vez

Ordenado: assegura que os dados são entregues aos seus receptores na ordem em que foram enviados

Desordenado: se é importante preservar a ordem, a aplicação é que deve detectar que os dados estão fora de ordem e, se necessário, reordená-los

• Ordem e Fidélidade não são ortogonais

Mensagem/Datagrama: o canal suporta o transporte de mensagens - sequências de bits processadas atomicamente

Fluxo: o canal não suporta mensagens - funciona como um "tubo" para uma sequência de bytes

Controlo de Fluxo: previne emissores "rápidos" de sobrecarregarem receptores "lentos" com dados

Número de Fins:

1. **Unicast/ ponto-a-Ponto:** só dois fins
2. **Broadcast:** todos os nós na rede
3. **Multicast:** subconjunto de nós na rede

Identificação: nome do processo e nome do canal

Protocolos Internet

APLICAÇÃO
TRANSPORTE
RÉDE
INTERFACE

- serviços específicos de comunicação
- comunicação entre dois (ou mais) processos
- comunicação entre dois computadores não diretamente ligados
- comunicação entre dois computadores diretamente ligados

UDP:

- Transporta mensagens
- send() e receive()
- os datagramas têm um tamanho máximo - desmontar e montar
- "connectionless"
- não fiável (perda e duplicação)
- Sem controlo de fluxo
- suporta multicast

TCP:

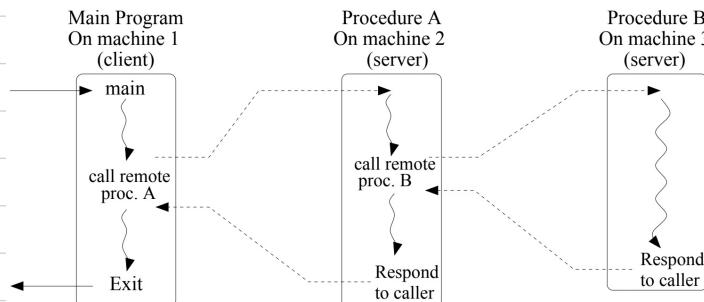
- canais de fluxo bidirecional
- write() e read()
- a aplicação deve preservar os limites das mensagens
- "connection-oriented"
- fiável (perda e duplicação)
- com controlo de fluxo
- só com dois endpoints (endereço IP, porta TCP)

Property	UDP	TCP
Abstraction	Message	Stream
Connection-based	N	Y
Reliability (loss & duplication)	N	Y
Order	N	Y
Flow control	N	Y
Number of recipients	1/n	1

Aplicação de Streaming: aplicação multimédia cujos conteúdos podem ser reproduzidos antes de serem completamente recebidos

Argumento Fim-a-Fim: "se tens de implementar uma função 'fim-a-fim', não a implementes nas camadas inferiores a não ser que melhore extremamente o desempenho"

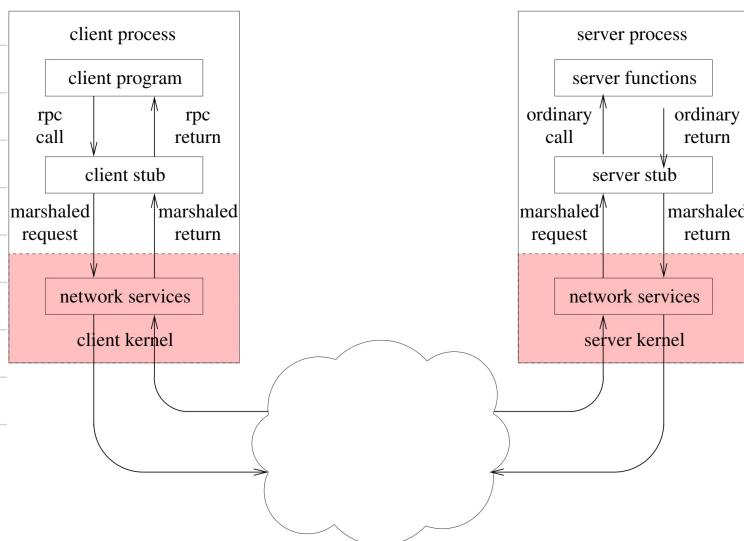
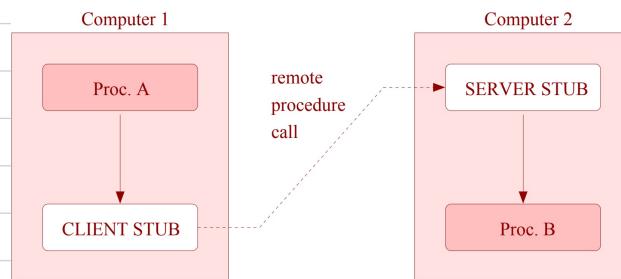
Chamada a Procedimentos Remotos



Client Stub: função local invocada pelo cliente

Server Stub: função local que invoca a função remota

As rotinas "stub" comunicam entre elas através da troca de mensagens



Client Stub:

Pedido:

1. Monta a mensagem - "marshalling" de parâmetros
2. Envia a mensagem ao servidor - write() / sendto()
3. Bloqueia à espera de resposta - read() / recvfrom()

Resposta:

1. Recebe resposta
2. Extrai os resultados - "unmarshalling"
3. Retorna ao cliente

Server Stub:

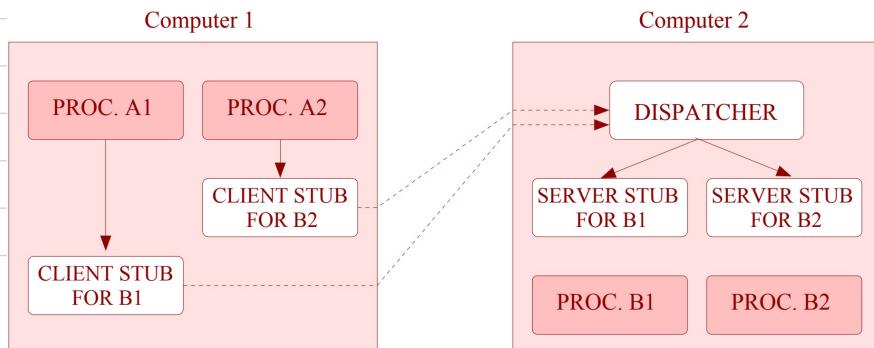
Pedido:

1. Recebe mensagem com pedido - read() / recvfrom()
2. Parsa a mensagem para determinar argumentos - "unmarshalling"
3. Chama a função

Resposta:

1. Monta mensagem com o valor de retorno da função
2. Envia a mensagem - write() / sendto()
3. Bloqueia à espera de um novo pedido

A identificação do procedimento é realizada pelo "dispatcher" - espaço de nomes hierárquico (serviço, procedimento)



Problemas:

1. Heterogeneidade de Plataformas → padronizar o formato **OU** recorre bem
2. Endereços como Argumentos → chamar por cópia/restaurar os parâmetros
3. Presença de Falhas: um cliente não consegue distinguir entre perda de um pedido, perda de uma resposta ou falha do servidor

PEDIDOS IDEMPOTENTES

- RPC "no mínimo uma vez": cliente reenvia pedido até receber resposta ou desistir
- retorna valor: RPC executou uma ou mais vezes
 - lança exceção: RPC executou várias vezes ou nenhuma vez

Resposta Perdida: se o procedimento não for idempotente, então a assinatura do RPC deve incluir o ID do pedido como argumento (gerido pelo Servidor) e o servidor deve manter uma tabela com as respostas previamente enviadas

Falha do Servidor: o servidor tem de relembrar o estado entre reinícios e, se os pedidos não forem idempotentes, relembrar também pedidos anteriores

PEDIDOS NÃO IDEMPOTENTES

- RPC "no máximo uma vez": cliente envia pedido uma vez
- retorna valor: RPC executou exatamente uma vez
 - lança exceção: RPC executou uma vez ou nenhuma vez

Com UDP, o middleware do RPC deve gerir o ID dos pedidos RPC e o server stub deve guardar uma tabela com as respostas

Falha do Servidor: o servidor tem de relembrar o estado entre reinícios

Falha do Cliente: o cliente pode não saber se tinha o lock - servidor tem de lidar

RPC "exatamente uma vez": virtualmente impossível no caso de ações extremas

Client	Strategy A → P			Strategy P → A			
	Reissue Strategy	APC	AC(P)	C(AP)	PAC	PC(A)	C(PA)
Always	Dup	OK	OK	OK	Dup	Dup	OK
Never	OK	Zero	Zero	Zero	OK	OK	Zero
When Ack	Dup	OK	OK	Zero	Dup	OK	Zero
When not Ack	OK	Zero	OK	OK	OK	Dup	OK

OK = Text printed once
Dup = Text printed twice
Zero = Text not printed at all

Clientes e Servidores

Concorrência: o objetivo é sobrepor I/O com processamento e tirar partido dos múltiplos "cores"

• A execução de um processo alterna entre intervalos de tempo de computação, pelo CPU, e intervalos de tempo de recesso, à espera de I/O

Servidor Iterativo: só tem uma thread e processa um pedido/concessão de cada vez, tendo cada passo/etapa uma operação que pode bloquear o servidor, impedindo-o de processar outros pedidos enquanto bloqueado

Servidor Multi-Thread: cada thread processa um pedido e, quando uma thread bloqueia com I/O, outra thread pode ser programada para correr no seu lugar

→ Uma thread "dispatcher" que aceita um pedido de concessão
→ Muitas threads "workers", em que cada uma processa todos os pedidos enviados no âmbito de uma única concessão

Servidor Guiado por Eventos: executa um ciclo em que espera por eventos (normalmente I/O) e processa esses eventos sequencialmente, um após o outro

• O bloqueio é evitado por operações de I/O não bloqueantes
• O ciclo entrega o evento ao FSM apropriado associado a cada pedido

CONCORRÊNCIA: THREAD VS. EVENTO

Concorrência baseada em Threads: só é necessário garantir isolamento no acesso a estruturas de dados partilhadas — pode usar só monitores e variáveis de condição, mas há implicações de modularidade e possibilidade de "deadlocks"

- Conforme o número de threads aumenta, o débito do sistema aumenta, depois nivela-se e finalmente decresce

Concorrência baseada em Eventos: é necessário gerir o estado explicitamente (com máquinas de estados) e partir o processamento de acordo com chamadas potencialmente bloqueantes — o problema é falta de atomicidade

- Conforme o número de pedidos na fila aumenta, o débito aumenta até chegar a um planalto

Threads ao nível do Kernel: implementadas diretamente pelo sistema operativo

- O escalonador do Kernel aloca "cores" às threads
- O SO mantém uma tabela de threads com informação de cada thread
- Todas as operações de gestão de threads envolvem chamadas ao sistema

Threads ao nível do Utilizador: implementadas por código ao nível do utilizador

- O Kernel não está ciente da existência de threads
- O SO não tem de suportar threads
- A biblioteca é responsável por gerir threads e mante uma tabela de threads
- Page-Fault de uma thread vai prevenir outras threads de executarem
- Não pode ser usado para explorar paralelismo em arquiteturas "multicore"

Implementação Híbrida: a biblioteca multiplexa threads ao nível do utilizador em threads ao nível do Kernel

Architecture	Paral.	I/O Oper.	Progr.
Iterative	No	Blocking	easy
Multi-threaded	Yes	Blocking	races
State-machine	Yes	Non-blocking	event-driven

Para tirar partido de múltiplos processadores / "cores" é necessário utilizar threads no núcleo do Kernel

↓ Threads não são tão geradas como processos
↓ Eventos limitam o número de threads

Problema: a execução da mesma tarefa em todos os pedidos de um cliente pode tacar o serviço desrespeitadamente

Solução: o serviço pode guardar algum estado (da sessão)

Falhas em Servidores "Stateful":

1. Perder o estado em quebras do servidor pode levar a:
 - a. ignorar ou rejeitar pedidos do cliente depois de recusar
 - b. interpretar mal pedidos do cliente enviados antes da quebra
2. Manter o estado em quebras do cliente pode levar a:
 - a. interpretar mal os pedidos enviados por outros clientes depois da quebra
 - b. esgotamento de recursos

Solução: o servidor empresta um recurso para o cliente por um intervalo finito de tempo; depois de expirar, o recurso pode ser libertado, a não ser que o cliente renove o seu empréstimo

Identificação de Clientes em Servidores "Stateful":

1. Usar o endereço do ponto de acesso, i.e., do "endpoint" do canal
2. Usar um "handle" independente da camada de transporte

Problema: É se o serviço mantém o estado na memória principal e o serviço responde ao equilíbrio de carga?

Solução: é necessário garantir que o estado é acessível a todos os servidores

Um servidor só pode ser "stateless" se cada mensagem do protocolo tem toda a informação para o seu processamento

Um servidor só pode ser "stateful" se cada mensagem do protocolo tem informação suficiente para se relacionar com a comunicação anterior

Desafios:

1. Componentes numa aplicação distribuída podem falhar, enquanto outros podem continuar a operar normalmente
2. Na Internet, é virtualmente impossível distinguir falhas da rede de falhas de ambiguidade ou até de um ambiente lento

Distribuição é mais difícil do que concorrência!
Sistemas distribuídos são inherentemente concorrentes

Sistemas Concorrentes: é necessário considerar todas as intercalações de execução
^{forníveis}

Sistemas Distribuídos: também é necessário considerar todas as forníveis falhas

Desafio: servidores executam com privilégios que os clientes normalmente não têm

Solução: autentica clientes e controla acesso a recursos e confidencialidade

Threads e Concorrência

Criação de Threads:

1. class ... implements Runnable → new Thread(new ...())
2. class ... extends Thread → new ...()
3. Runnable var = () → ... → new Thread(var)
4. Thread.ofVirtual().start() → ...
5. Executors.new...() → executor.execute(...) → executor.shutdown()

Métodos:

1. start(): as novas threads têm visibilidade nos dados da thread precedente
2. join(): as threads obtêm visibilidade nos dados da thread que terminou
3. interrupt()

• Erros de consistência de memória ocorrem quando threads diferentes têm visões inconsistentes do que devem ser os mesmos dados

• Escrever sob um "lock" e ler depois de adquirir o mesmo "lock" garante visibilidade

Locks:

1. synchronized ... () { ... }
2. synchronized(this) { ... }
3. Object lock = new Object(); → synchronized(lock) { ... }
4. ReentrantLock l = new ReentrantLock(); → l.lock(); ...; l.unlock();
5. while (!lock) { try { wait(); } catch {} } → lock = true; notifyAll()

Deadlock: se a transferência de "locks" cria ciclos

Barrier: mecanismo de apoio à sincronização que permite que um conjunto de threads esperem todas juntas pelas ondas para alcançarem um ponto comum da barreira — Barrier b = new Barrier(N); new Thread(new Client(b, i)).start()

↑
número de threads

↑
número da thread

→ b.barrier()

A ordem de aquisição dos "locks" é relevante

↓
É possível ter uma estrutura mutável que armazene células com "locks" internos, em que o "lock" da estrutura maior só deve ser mantido por pouco tempo

É possível ter múltiplas variáveis de condição

```
class Barrier {  
    final int N; int proc = 0; int stage = 0;  
  
    public Barrier(int n) { this.N = n; }  
  
    public synchronized void barrier() throws  
        InterruptedException {  
        int local = stage;  
        proc++;  
        if (proc < N) {  
            while (local == stage)  
                wait();  
        } else {  
            stage++;  
            proc = 0;  
            notifyAll();  
        }  
    }  
}
```

Causalidade e Tempo Lógico

O problema com o tempo é que não é universal

Propriedades do Tempo:

1. O tempo precisa de memória
2. O tempo é local
3. A sincronização do tempo é mais difícil à distância
4. O tempo é um mau representante da causalidade

- A causalidade é uma relação de ordem parcial
- A causalidade é só influência potencial

Como registrar a causalidade? Históricos Causais ou Relógio Vetorial
→ etiquetar unicamente cada evento

Históricos Causais: colecionar memórias como conjuntos de eventos únicos - a inclusão de conjuntos (\subseteq) explica a causalidade

- "Tu estás no seu passado se eu sei o seu histórico"
- "Se nós não conhecemos os históricos um do outro, então nós somos concorrentes"
- "Se os nossos históricos não os nemos, nós somos o mesmo"

Ex: $\diamond = \{a_1, a_2\}$ $\star = \{a_1, a_2, b_1, b_2\}$ - $\diamond \rightarrow \star \Leftrightarrow \{a_1, a_2\} \subset \{a_1, a_2, b_1, b_2\}$

Nota: $\{e_n\} \subset C_x \Rightarrow \{e_1, \dots, e_n\} \subset C_x$

$$\{a_1, a_2, b_1, b_2, b_3, c_1, c_2, c_3\} \Leftrightarrow \{a \rightarrow 2, b \rightarrow 3, c \rightarrow 3\} \Leftrightarrow [2, 3, 3]$$

Relógios Virtuais: vetores que armazem um número fixo de processos com identificações totalmente ordenadas — a união de conjuntos torna-se \sqcup (máximo ponto-a-ponto)

Ex: $\diamond = [2, 0, 0]$ $\star = [2, 2, 0]$ — $\diamond \rightarrow \star \Leftrightarrow 2 \leq 2 \wedge 0 \leq 2 \wedge 0 \leq 0$

Comparar vetores é linear no tamanho do vetor, mas pode ser melhorado ao registrar o último evento — o passado causal exclui o próprio evento

Ex: $[2, 0, 0] \oplus [1, 0, 0]_{a_2}$
 $\downarrow [1, 0, 0]_{a_2} \rightarrow [2, 1, 0]_{b_2}$ se e só se "dot a_2 index 2" ≤ 2 ?

Nem sempre é importante registrar todos os eventos, mas apenas os eventos de atualização em réplicas de dados

- **EVENTO RELEVANTE** — etiqueta única e adicionada ao histórico
- **EVENTO IRRELEVANTE** — não etiquetado nem adicionado ao histórico

Versões podem ser colecionadas e unidas mais tarde
Históricos causais só são unidos sob a união de versões nem novo

- A causalidade é importante porque o tempo é limitado
- A causalidade é sobre memória de eventos relevantes
- Históricos Causais são representações muito simples de causalidade
- Relógios Virtuais e Vetores de Versões codificam eficientemente Históricos Causais
- Representações gráficas permitem visualização de causalidade
- Todos os mecanismos são codificações de Históricos Causais

Tolerância a Falhas

Falha: quando um sistema/componente não se comporta de acordo com a sua especificação

Tolerante a Falhas: um sistema que se comporta corretamente apesar da falha de alguns dos seus componentes — redundância

Redundância Modular Tripla: cada nó é triplicado e trabalha em paralelo — o output de cada módulo é ligado a um elemento de voto (também triplicado) cujo output é a maioria dos seus inputs

• A não ser que um sistema distribuído seja tolerante a falhas, vai ser menos fiável do que um sistema não distribuído

• A redundância de hardware inherentemente a um sistema distribuído torna-o particularmente apropriado para o tornar tolerante a falhas

Fiabilidade $R(t)$: probabilidade de um sistema não ter falhado até ao tempo t

M_{TTF}: tempo médio para falhar

M_{TR}: tempo médio para reparar

$$\text{Disponibilidade: } \alpha = \frac{\text{M}_{\text{TTF}}}{\text{M}_{\text{TTF}} + \text{M}_{\text{TR}}}$$

• Assume que um sistema deve ser reparado depois de falhar

Modelo de Sistema Distribuído: um conjunto de processos sequenciais que executam os passos de um algoritmo distribuído

- Os processos comunicam e sincronizam através da troca de mensagens
- Os processos podem ter acesso a um relógio local
- O sistema distribuído pode ter modelos de falhas parciais

Modelos de Sincronia: caracterizam o sistema de acordo com o comportamento temporal dos seus componentes (processo, relógios locais e canais de comunicação)

Síncronos: se e só se existem limites conhecidos para:

1. o tempo que um processo demora a executar um passo
2. a desviação do tempo dos relógios locais
3. atrasos de mensagens

Assíncronos: não fazem previsões acerca do comportamento temporal do sistema distribuído

Modelos de Falha: caracterizam o sistema de acordo com os tipos de falha que os seus componentes podem exibir

1. Quebra: um componente comporta-se corretamente até algum instante de tempo, a partir do qual deixa de responder a qualquer input
2. Omissão: um componente não responde a alguns dos seus inputs
3. Timing/Desempenho: um componente não responde a tempo (cedo ou tarde)
4. Byzantino/Arbitrário: um componente comporta-se de forma totalmente anárquica

Queda - Recuperação: assume-se que um processo faltoso pode quebrar e recuperar indefinidamente, i.e., um número ilimitado de vezes



Compromisso Atómico

Problema: Como garantir que num conjunto de operações, executadas em diferentes processadores, ou todas são executadas ("committed") ou nenhuma é executada ("abortadas")?

Uma transação tem de ter as propriedades ACID

Compromisso Atómico: considere-se um conjunto de N processos tal que:

1. Cada processo tem de decidir um de dois valores: "commit"/abortar
2. Cada processo deve votar/propor um desses dois valores
3. O valor decidido por cada processo deve satisfazer as seguintes condições:
 - a. Todos os processos que decidem devem decidir o mesmo valor
 - b. A decisão de um processo é final e inalterável
 - c. Se algum processo decide "commit", então todos os processos devem ter votado "commit"
 - d. Se todos os processos votaram "commit" e não existem falhas, então todos os processos devem decidir "commit"
 - e. Para qualquer execução que contenha algumas falhas que o algoritmo está projetado para tolerar, em qualquer ponto dessa execução, se todas as falhas existentes estão reparadas e não ocorrem novas falhas por um período de tempo suficientemente longo, então todos os processos eventualmente chegam a uma decisão

Propriedade "Safety": algo (mau) não vai acontecer

Propriedade "Diverges": algo (bom) deve acontecer

COMMIT EM DUAS FASES

Assumções:

1. Os processos podem falhar ao querer e recuperar
2. Cada processo tem armazenamento cujo conteúdo sobrevive a uma quebra

Coordenador: em cada momento, só há um processo coordenador

Participante: processo que realiza uma operação

Fases:

1. A pedido da aplicação:

C: envia um VOTE-REQUEST para cada processo e espera pelas respostas

P: ao receber um VOTE-REQUEST, responde VOTE-COMMIT ou VOTE-ABORT

2. Quando o coordenador determina que é altura de decidir:

C: decide/envia GLOBAL-COMMIT ou GLOBAL-ABORT

P: decide de acordo com a mensagem recebida do coordenador

Timeout: só ocorre quando um processo está à espera de algum evento

Ações de Timeout: ações tomadas por um processo após um timeout

C: só espera por mensagens no estado WAIT → decide abortar e envia

GLOBAL-ABORT a todos os participantes

P:

INIT → decide abortar e mover-se para o estado correspondente

READY → deve executar um protocolo de término para descobrir o desfecho

Protocolo de Término: o participante deve comunicar com os outros participantes para descobrir o desfecho - se algum participante vota a decisão (votou VOTE-ABORT ou recebeu GLOBAL-*)

Ações de Recuperação: ações tomadas por um processo depois de recuperar de uma quebra

↓
· Se o processo ainda não decidiu, então:

1. Se quebrou enquanto esperava por uma mensagem, toma a ação de time-out corrigido
2. Senão (o coordenador está em INIT), decide ABORT

· Para permitir recuperação, os processos devem escrever o estado do protocolo como entradas para um registo ("log") no armazenamento estável

Problema: este protocolo pode precisar que os clientes bloqueiem (esperem mais do que um timeout de comunicação)

Impossibilidade de Recuperação Independente: nenhum protocolo permite sempre recuperação local, i. e., sem comunicação com outros processos

Impossibilidade de Não-Bloqueio: nenhum protocolo nunca bloqueia na presença de falhas de comunicação ou falhas em todos os outros processos

COMMIT EM TRÊS FASES

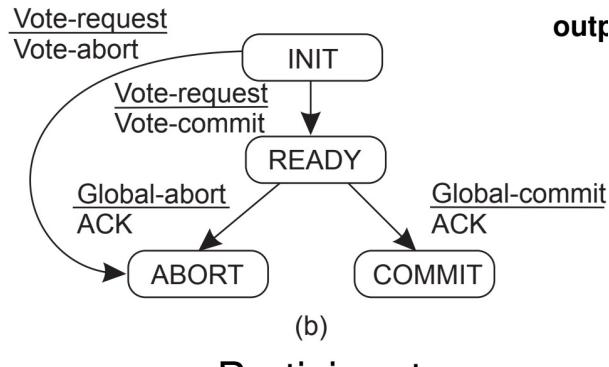
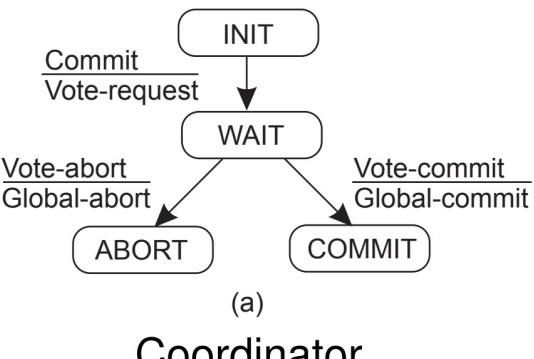
· Adiciona uma fase entre as duas fases do protocolo anterior, em que o coordenador revela a sua intenção de COMMIT

Estado PRECOMMIT: o processo vai decidir COMMIT a não ser que o coordenador falhe — garante a condição de não bloqueio porque nenhum processo pode fazer COMMIT enquanto outro processo estiver num estado incerto (INIT, WAIT, READY), i. e., poder decidir COMMIT ou ABORT

A NÃO SER QUE

1. todos os processos falham
2. não existe maioria

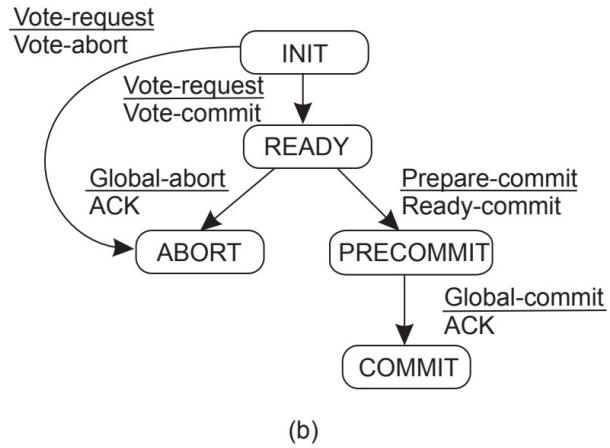
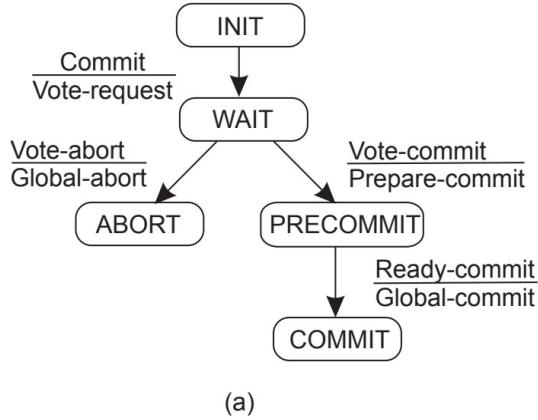
COMMIT EM DUAS FASES



Coordinator

Participant

COMMIT EM TRÊS FASES



Coordinator

Participant

Elições

Porquê? Muitos algoritmos baseiam-se num processo com um papel especial
O quê? Quando completado o algoritmo, todos os nós nem falha concordam
em quem é o coordenador/líder

Algoritmos Garcia-Molina: é possível garantir tolerância a falhas através
de duas abordagens

1. Mascarar Falhas: usar algoritmos que continuam a funcionar corretamente, mesmo
se algum componente do sistema falhar.
2. Reorganizar o Sistema: interromper a operação normal e gastar algum tempo
a reorganizar o sistema

Assumções:

1. Todos os nós cooperam e usam o mesmo algoritmo
4. Todos os nós têm algum armazenamento estável / Registo
5. Quando um nó falha, interrompe imediatamente o processamento
3. O subsistema de comunicação não gera mensagens espontaneamente
6. Não há erros de transmissão (mas as mensagens podem ser perdidas)
7. As mensagens são entregues na ordem em que são enviadas
8. O sistema de comunicação não falha e tem um limite superior no tempo
para entregar uma mensagem, T
9. Um nó responde sempre a mensagens recebidas, sem atraso

S(i).s: estado do nó i - DOWN, ELECTION, NORMAL

S(i).c: o coordenador de acordo com o nó i

Asserções:

1. Em qualquer instante de tempo, para quaisquer dois nós, se ambos estiverem no estado normal, então ambos concordam no coordenador.
 $\rightarrow \forall i, j : S(i).s = S(j).s = \text{NORMAL} \rightarrow S(i).c = S(j).c$
2. Se não ocorrerem falhas durante a eleição, então o protocolo vai eventualmente transformar o sistema em qualquer estado para um estado em que:
 - a. todos os nós não-falhos, tenham $S(i).s = \text{NORMAL}$
 - b. exista um nó não-falho i tal que $S(i).c = i$

ALGORITMO DO BULLY

Convenção: um nó é mais forte quanto menor for o seu identificador

Fases:

1. Um nó que se quer tornar um líder verifica se nós mais fortes estão presentes ao enviá-los uma mensagem ARE-U-THERE
 - a. Se presente, um nó mais forte responde com uma mensagem YES e inicia uma nova eleição
 - b. Um candidato cujo desafio tenha sido respondido derrete
2. Se o candidato (nó i), não receberem respostas ao fim de $2T$
 - a. Envia uma mensagem HALT para nós mais fracos, que, ao receberem:
 - i. Definem o seu estado para ELECTION
 - ii. Cancelam qualquer eleição que já tenha começado
 - b. 1 unidade de tempo depois, o nó i envia uma mensagem NEW-LEADER aos nós mais fracos e todos os nós (o candidato ao enviar e os outros ao receberem) definem $S(j).c = i$ e $S(j).s = \text{NORMAL}$

Falhas?

Líder: um processo inicia uma nova eleição

Candidato: um processo inicia uma nova eleição

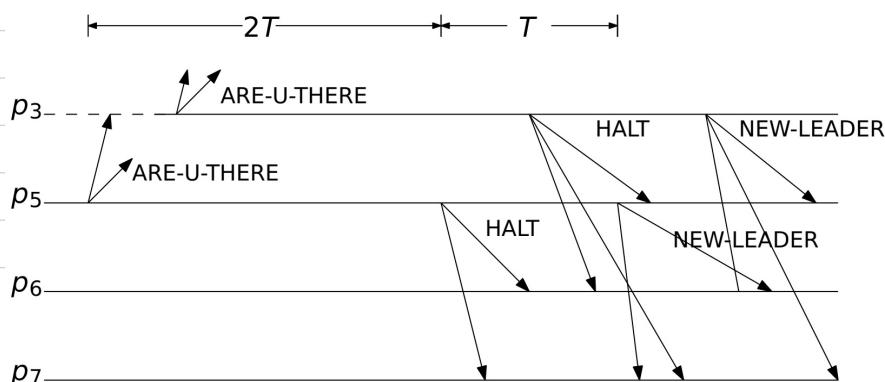
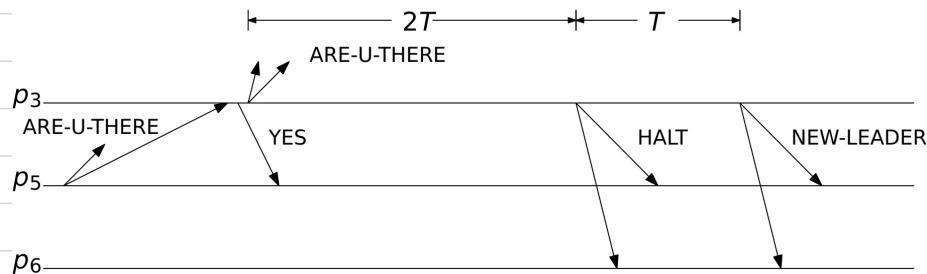
Outro: não interessa

Recuperação? O nó inicia uma eleição

• Caso receba mais do que um HALT, um nó atualiza sempre o candidato para o mais forte

• Se o divisor não coincide com o candidato, o nó descarta a mensagem NEW-LEADER

• Se não estiver no estado ELECTION, o nó descarta a mensagem NEW-LEADER



Comunicação Baseada em Grupos

"Primary Backup Replication":

- Um servidor é o primário e os restantes são backups
- Os clientes afirmam pedidos para o servidor primário
- O servidor primário executa os pedidos, atualiza o estado dos backups e responde aos clientes depois de receber ACKs suficientes dos backups
- Essencialmente, o primário ordena os diferentes pedidos dos clientes
- Se o primário falha, ocorre um failover e um dos backups torna-se o novo primário

Primário falha depois de enviar resposta ao cliente: transparente para o cliente, a não ser que a mensagem de resposta seja perdida e o primário falhe antes de a retransmitir

Primário falha antes de enviar atualização aos backups: nenhum backup recebe a atualização e, se o cliente retransmite o pedido, vai ser lidado como um novo pedido pelo novo primário

Primário falha depois de enviar a atualização e antes de enviar a resposta: se o cliente retransmitir o pedido, o novo primário vai responder, pelo que a mensagem de atualização deve incluir a resposta se a operação não for idempotente
— a atualização deve ser entregue atomicamente!

Qual é a tolerância a falhas? depende do modelo

1. Quórum: $n - 1$ respostas faltosas

2. Democracia: é necessária uma maioria para prever a existência de mais de que um primário ao mesmo tempo

Problema: quando um backup recupera, o seu estado está obsoleto, pelo que não pode aplicar as atualizações nem enviar ACKs ao novo primário

Solução: usar um protocolo de transferência de estado para sincronizar o estado do backup com o do primário

reenviar atualizações em falta
transfere o próprio estado

Problema: esperar por ACKs dos backups aumenta a latência

Solução: o primário pode enviar a resposta ao cliente antes de receber ACKs dos backups

