Computação Paralela: Algoritmos e Aplicações

Prof. Amit Bhaya,
Programa de Engenharia Elétrica, COPPE/UFRJ
15/05/2001 -- 18/05/2001
http://www.nacad.ufrj.br/~amit/cpaa2001.html
NACAD = Núcleo de Computação de Alto Desempenho

Conteúdo do minicurso

- Overview de Computação Paralela
- Projeto de Algoritmos Paralelos
- Modelagem de Desempenho Paralelo
- Redes de Interconexão
- Comunicação entre processadores
- Paradigmas de programação paralela
- MPI (Message Passing Interface)
- Produtos de vetores e matrizes
- Fatorações LU e Cholesky
- Sistemas triangulares e tridiagonais
- Métodos iterativos
- Assincronismo
- Fatoração QR
- Problemas de Autovalor
- FFT
- Outras aplicações

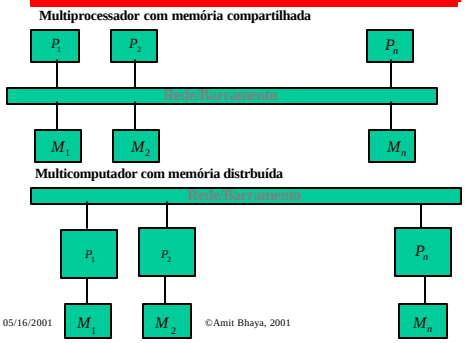
Motivos para cautela

- Pouco software disponível
- Ambiente de computação não consolidado (não há muitas ferramentas de debugging , visualização etc.)
- Mercado comercial instável (fabricantes aparecem e desaparecem rapidamente)

Porque paralelismo?

- Limites fundamentais na velocidade de um único processador
- Throughput (produtividade líquida) apresenta alta razão benefício/custo)
- É possível aproveitar recursos existentes (clusters de PCs)

Arquiteturas paralelas básicas



Aspectos de arquitetura paralela

- Mecanismo de controle: SIMD vs MIMD
- Operação: síncrono vs. Assíncrono
- Organização de memória: privativa vs. Compartilhada
- Espaço de endereçamento: local vs. Global
- Acesso a memoria: uniforme vs não uniforme
- Granularidade: potência de processadores individuais
- Topologia da rede de interconexão

Compromissos entre arquiteturas

	MemóriaCompartilhada	Memória Distribuída
Programabilidade	Mais fácil	Mais difícil
Escalabilidade	Mais difícil	Mais fácil

Categorias de máquinas paralelas

- Processador vetorial ou array
- SMP: multiprocessador simétrico
- MPP: processador maçicamente paralelo
- DSM: memória compartilhada (shared) e distribuída
- Cluster de PCs ou estações em rede
- Híbridos e combinações
  - -- SMP ou MPP com processadores vetoriais
  - -- Clusters de SMPs em rede, ...

Exemplos (1/2)

Processadores vetoriais ou array

•Supercomputadores vetoriais: CRAY, CDC, ETA

•Minisupers: Convex, Alliant

•Processadores array: FPS

SMP

•Primeiros modelos: Sequent, Encore

•Modelos atuais: HP, IBM, SGI, Sun, PCs

O NACAD possui CRAY, SP-2 (IBM), e cluster de PCs

Exemplos (2/2)

MPP

•Primeiros modelos: Ncube, Intel iPSC

•SIMD: TMC CM-1, CM-2, MassPar

•Modelos mais recentes: Intel Paragon, IBM SP, Cray T3D/E

MPP vetorial

•Intel iPSC/2, FPS T-series, TMC CM-5

DSM

•KSR, Convex Exemplar, SGI Origin

Hierarquia de memória

Arquiteturas de máquinas de alto desempenho, mesmo com um processador, são classsficadas de acordo com uma *hierarquia de memória:*

•Registros
•Cache(s) on-chip
•Cache(s) off-chip
•Memória de acesso randômico (RAM)
•Memória remota (off-processor)
•Memória virtual (paginação)
•Armazenamento secundário (discos)
•Armazenamento terciário (fitas)

Localidade (localização) de dados e reutilização são críticos para alto desempenho

Paradigmas de programação paralela

•Linguagens funcionais (dataflow)

•Compiladores paralelizadoras (baseadas em malhas com diretivas)

•Paralelismo em dados (operações simultâneas nos elementos do array)

•Memória compartilhada (múltiplos fios [threads] executando pool de tarefas comuns)

•Acesso a memória remota (comunicação unilateral entre processos: put/get)

•Troca de mensagens [Message passing] (comunicação bilateral entre processos: send/recv)

Linguagens e padrões

•Paralelo em dados: F90, HPF

•Troca de mensagens: MPI, PVM

•Memória compartilhada: pthreads, OpenMP

•Álgebra linear: BLAS, PBLAS,BLACS

(HPF = High Performance FORTRAN)

(PVM = Parallel Virtual Machine)

(BLAS = Basic Linear Algebra Subroutines)

Projeto de Algoritmos Paralelos

•Decompor problemas em tarefas de granularidade fina para maximizar paralelismo potencial

•Determinar padrões de comunicação entre tarefas

•Combinar em tarefas de granularidade mais grossa, se necessário, para reduzir custos de comunição etc.

•Alocar tarefas a processadores, considerando os compromissos entre comunicação e concorrência

Paradigmas algoritmicos

•Decomposição em domínio (baseada nos dados)

•Decomposição funcional (baseado na computação)

•Paralelismo embaraçoso (tarefas independentes ou desacopladas

•Paralelismo nos dados (operações em arrays)

•Dividir-para-conquistar (tipo árvore)

•Pipeline (sobreposição entre etapas)

Aspectos de comunicação

•Latência e largura de banda

•Roteamento

•Padrões globais

-- broadcast

-- redução

-- todos-a-todos

•Contenda, largura de faixa agregada

Alocação de tarefas/dados a processadores

- Particionamento
- Granularidade
- Mapeamento
- Scheduling
- Balanceamento de carga

Fatores que afetam desempenho

- Balanceamento equitativo de carga
- Concorrência: execução simultânea de tarefas
- Overhead: tarefas ausentes em computação sequencial
  - -- comunicação
  - -- sincronização
  - -- tarefas redundantes (ociosidade)
  - -- tarefas especulativas

Modelos de computação paralela

Modelos teóricos de computação paralela incluem:

- PRAM – Parallel Random Access Machine
- LogP – Latency/Overhead/Gap/Processors
- BSP – Bulk Synchronous Parallelism
- CSP – Communicating Sequential Processes

E muitas outras ...

Referências

G. S. Almasi & A. Gottlieb, ***Highly Parallel Computing***, 2nd ed., Benjamin/Cummings, 1994

D. E. Culler, J. P. Singh & A. Gupta, ***Parallel Computer Architecture,*** Morgan Kaufmann, 1998

H. El-Rewini & T. G. Lewis, ***Distributed and Parallel Computing***, Manning, 1998

I. T. Foster, ***Designing and Building Parallel Programs***, Addison-Wesley, 1995

M. J. Quinn, ***Parallel Computing: Theory and Practice***, McGraw-Hill, 1994

A. Y. Zomaya,editor, ***Parallel and Distributed Computing Handbook***, McGraw-Hill, 1996

Projeto de algoritmos paralelos

Modelo de programação paralela

- ***Computação paralela*: execução simultânea de 2 ou mais *tarefas***
- ***Tarefa* encapsula programa sequencial e memória local**
- **2 tarefas podem ser conectadas por *canal***
- ***Send* é assíncrono: tarefa que manda retorna a execução imediatamente**
- ***Receive* é síncrono: execução de tarefa receptora bloqueada até a mensagem ficar disponível**
- **Tarefas podem ser mapeadas a processadores em diversas maneiras, incluindo múltiplas tarefas por processador**

Implicações do modelo

- **Semântica do programa não depende do mapeamento tarefa a processador**
- **Desempenho sensível ao mapeamento. Motivos: balanceamento de carga, concorrência e comunicação**
- **Canais de comunicação podem ou não refletir rede de interconexão subjacente (por exemplo, duas tarefas comunicantes podem estar alocados ao mesmo processador, ou a processadores diferentes porém fisicamente conectados, ou a 2 procs. que não são conectados, implicando em roteamento de mensagens)**
- **Programas paralelas devem ser escaláveis (i.e., executam corretamente independente do número de processadores disponíveis)**

Exemplo: Jacobi para Eq de Laplace (1-D)

Equação de Laplace em uma dimensão:

$y''(t) = 0$, $a \leq t \leq b$, com condições de fronteira

$y(a) = \alpha$, $y(b) = \beta$

Aproximação a diferenças finitas:

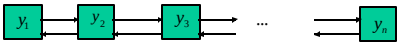
$$y_{i+1} - 2y_i + y_{i-1} = 0, i = 1,\ldots,n, y_0 = \alpha, y(b) = \beta$$

Iteração tipo Jacobi

$$y_i^{(k+1)} = \frac{y_{i-1}^{(k)} + y_{i+1}^{(k)}}{2}, \; i = 1,\ldots, \; n$$

Exemplo: Jacobi para Eq de Laplace (1-D)

Define *n* tarefas, um para cada y_i



Inicializa y_i

Para $k = 1$

send y_i tarefa $i-1$

send y_i tarefa $i+1$

recv y_{i+1} de tarefa $i+1$

recv y_{i+1} de tarefa $i-1$

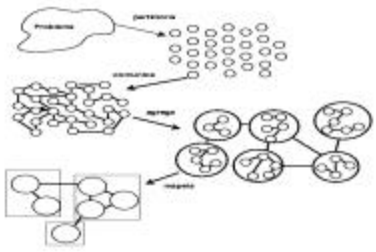
$y_i = (y_{i-1} + y_{i+1})/2$

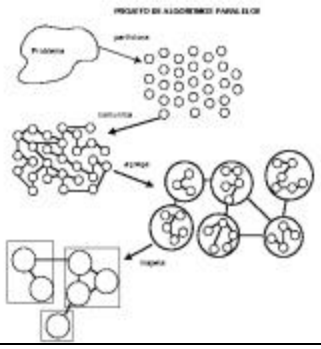
End

Projeto de algoritmos paralelos

- ***Particionamento:*** **Decompor problema em tarefas de granularidade fina para maximizar paralelismo potencial**
- ***Comunicação:*** **Determinar padrão de comunicação entre tarefas**
- ***Agregação:*** **Combinar em tarefas de granularidade grossa, se necessário, para reduzir custos de comunicação e outros**
- ***Mapeamento:*** **Alocar tarefas a processadores, sujeito ao s compromissos entre paralelismo (puro) e custo de comunicação.**

Projeto de algoritmos paralelos





Tipos de particionamento

•**Decomposição de *Domínio:* particionamento dos dados**

ex.: pontos de uma malha (grid) em dimensão 2, 3, …

• **Decomposição *funcional:* particionamento da computação**

ex.: Componentes em modelo climático (atmosfera, oceano, terra, etc.)

Decomposição de domínio

Cuidados com Particionamento

•Identificar pelo menos uma ordem de grandeza a mais de tarefas d o que número de processadores disponíveis na máquina

•Evitar cálculos ou armazenamento redundantes

•Criar tarefas de tamanho tão uniforme quanto possível

•Número de tarefas, ao invés do tamanho da tarefa, deve aumentar em função da dimensão do problema

Tipos de comunicação

•Local versus global

•Estruturada versus não estruturada

•Estática versus dinâmica

•Síncrona versus assíncrona

Cuidados com comunicação

•Frequência e volume de comunicação deve ser razoavelmente uniforme ao longo das tarefas

•Comunicação deve ser tão localizada quanto possível

•Comunicação deve ser concorrente (simultânea)

•Comunicação não deve inibir execução concorrente de tarefas

•Sobreposição (overlapping) entre comunicação e computação pode melhorar desempenho, sempre que viável

Agregação

•Aumento no tamanho da tarefa reduz comunicação porém também reduz concorrência potencial e flexibilidade

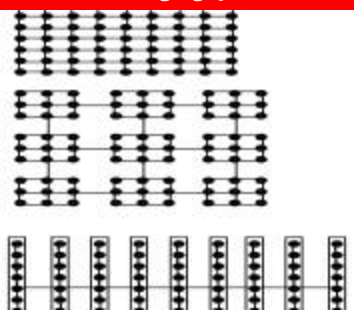
•Comunicação é proporcional à área de superfície do subdomínio, ao passo que computação é proporcional ao seu volume

•Decomposições de dimensão maior possuem razão superfície-volume mais favoráveis

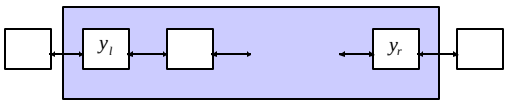
•Comunicação pode (as vezes) ser evitada através de replicação de computação em várias tarefas

•Subtarefas que não podem ser executadas concorrentemente são candidatas para serem agregadas em uma única tarefa

Agregação



Exemplo: Agregação



EXEMPLO: AGREGAÇÃO

Programa para tarefa i

Inicialize $y_l, \ldots, y_r$
for $k = 1, \ldots$
send y_l a tarefa $i-1$
send y_r a tarefa $i+1$
recv y_{r+1} de tarefa $i+1$
recv y_{l-1} de tarefa $i-1$
  for $j = l$ **to** r
  $\bar{y}_j = (y_{j-1} + y_{j+1})/2$
  end $y = \bar{y}$
end

EXEMPLO: SOBREPOSIÇÃO DE COMUNICAÇÃO E COMPUTAÇÃO

Programa para tarefa i
Inicialize $y_l, \ldots, y_r$
for $k = 1, \ldots$
  send y_l a tarefa $i-1$
  send y_r a tarefa $i+1$
  for $j = l+1$ **to** $r-1$
  $\bar{y}_j = (y_{j-1} + y_{j+1})/2$
  end
  recv y_{r+1} de tarefa $i+1$
  $\bar{y}_r = (y_{r-1} + y_{r+1})/2$
  recv y_{l+1} de tarefa $i-1$
  $\bar{y}_l = (y_{l-1} + y_{l+1})/2$
  $y = \bar{y}$
end

Mapeamento

Duas estratégias básicas para a alocação de tarefas a processadores:

•Aloque tarefas que podem executar concorrentemente a processadores diferentes

•Aloque tarefas que comunicam frequentemente no mesmo processador

Problema: Estas duas estratégias frequentemente conflitam

Em geral, a solução exata a este compromisso é um problema da class NP-completo, de modo que heurística é utilizada para achar uma solução razoável

Existem estratégias estáticas e dinâmicas

Balanceamento de carga

•Biseção recursiva

•Algoritmos locais

•Métodos probabilísticos

•Mapeamentos cíclicos

Scheduling de tarefas

- Gerente/operário
- Gerente hierárquico/operário
- Esquemas descentralizados
- Deteção de terminação

Exemplo: Modelo da atmósfera

Particionamento

$n_x \times n_y \times n_z$ pontos de malha em modelo 3-D de diferenças finitas

Tipicamente gera 10^5 a 10^7 tarefas.

Comunicação

- Molécula computacional de 9 pontos na direção horizontal e de 3 pontos na vertical
- Computação de física nas colunas verticais
- Operações globais para determinar massa total

Exemplo: Modelo da atmósfera

Agregação

- Quatro pontos da malha por tarefa horizontalmente reduz comunicação a vizinhos mais próximos
- Coluna vertical inteira por tarefa elimina comunicação para cálculos de física

Gera $n_x n_y / 4$ tarefas, tipicamente mil a cem mil

Mapeamento

Mapeamento cíclico reduz desbalanceamento gerado pelo cáclulos de física

Exemplo: Modelo da atmósfera

Referências

K. M. Chandy & J. Misra, ***Parallel Program Design: A Foundation***, Addison-Wesley, 1988

I. T. Foster, ***Designing and Building Parallel Programs*** , Addison-Wesley, 1995

V. Kumar, A. Grama, A. Gupta & G. Karypis, ***Introduction to Parallel Computing: Design and Analysis of Algorithms*** , Benjamin/Cummings, 1994

M. J. Quinn, ***Parallel Computing: Theory and Practice***, McGraw-Hill, 1994

Modelagem de desempenho

Enfoques para avaliação de desempenho

- Análise de escalabilidade
- Extrapolação a partir de observações
- Análise assintótica
- Modelagem realística

Lei de Amdahl

Hipóteses: fração serial = s, $0 \leq s \leq 1$, fração p-paralela = $1 - s$

Portanto,

$$T_p = sT_1 + (1-s)T_1 / p,$$
$$S_p = p/(sp + (1-s)),$$
$$E_p = 1/(sp + (1-s)).$$

Corolário: $S_p \to 1/s$ e $E_p \to 0$ quando $p \to \infty$

P. ex., se $s = 0.1$, então speedup máximo possível = 10, qq. q. seja p.

Este resultado ocasionou pessimismo nos primórdios de computação paralela.

Medidas de desempenho paralelo

T_1 = tempo de execução serial em um processador

T_p = tempo de execução paralelo em p processadores

Speedup $S_p = T_1 / T_p$

Eficiência $E_p = T_1 / (pT_p)$

Portanto $E_p = S_p / p$ e $S_p = pE_p$

Pseudoteorema: $S_p \leq p$ e $E_p \leq 1$

Porém, anomalias de speedup ocorrem na prática, p. ex., em função de mais recursos (e.g. cache) quando p aumenta.

Escalamento do problema

Lei de Amdahl é relevante somente quando o problema é fixo, ou quando a fração serial independe do tamanho do problema, o que se verifica raramente.

Computadores maiores são utilizados para resolver problemas maiores, e fração serial geralmente diminui quando o tamanho do problema aumenta

Taxa de aumento do problema pode ser caracterizado pela manutenção de alguma grandeza invariante enquanto o número de processsadores varia. Candidatos plausíveis incluem

- Tamanho total do problema [Amdahl]
- Trabalho por processador [Gustafson]
- Tempo total de execução [Worley]
- Memória por processador [Sun]
- Eficiência [Grama]
- Erro computacional [Singh]

Escalabilidade

Escalabilidade se refere a eficácia de um algoritmo paralelo na utilização de processadores adicionais.

Um algoritmo é denominado ***escalável*** em função do aumento do número de processadores se sua eficiência pode ser mantida constante (ou no mínimo limitada acima de zero) pelo aumento do tamanho do problema.

Um algoritmo escalável neste sentido poderia no entanto não ser prático se a taxa de aumento do tamanho do problema resulta em tempo total de execução inaceitável.

Perigos de extrapolação

Considere três algoritmos hipotéticos para problemas de tamanho n cujo custo serial é $n + n^2$:

1. $T_p = n + n^2 / p$

$T_p = (n + n^2) / p + 100$

$T_p = n + n^2 / p + 0.6p^2$

Para os três algoritmos $S_p \approx 10.8$ quando e

Porém o comportamento de cada um é bastante diferente para n e p maiores.

Perigos de análise assintótica

Análise assintótica é frequentemente baseada em um modelo irrealista de computação paralela (e.g. PRAM).

Estimativas assintóticas se aplicam para n e p grandes, porém podem ser irrelevantes para os valores de n e p de interesse prático.

Termos de ordem mais baixa podem ser significativas para os valores de n e p de interesse prático.

Exemplo: Se complexidade for $10n + n \log n$, termo linear é maior para $n < 1024$.

Constantes de proporcionalidade podem ser decisivas na prática.

Exemplo: Complexidade $10n^2$ melhor do que $1000n \log n$, para $n < 996$.

Modelagem de desempenho paralelo

Tempo de execução é tempo decorrido entre o instante quando o primeiro processador começa execução até o instante quando o último termina execução.

Em qualquer instante durante execução, cada processador está computando, comunicando ou ocioso.

Portanto, tempo total de execução no processador j é dado por:

$$T_{comp}^{j} + T_{comu}^{j} + T_{ocio}^{j}$$

É frequentemente mais fácil determinar tempo médio de execução por processador

$$T_p = \frac{1}{p}\left(T_{comp} + T_{comu} + T_{ocio}\right)$$

$$= \frac{1}{p}\left(\sum_{j=1}^{p} T_{comp}^{j} + \sum_{j=1}^{p} T_{comu}^{j} + \sum_{j=1}^{p} T_{ocio}^{j}\right)$$

Tempo de computação

Tempo de computação é o tempo de execução serial mais o tempo gasto em qualquer computação adicional de execução paralela.

Taxa de computação pode variar em função do tamanho de problema por causa de efeitos de cache, assincronismo, etc.

Tempo de comunicação

Tempo de comunicação é o tempo gasto enviando e recebendo mensagens.

Tempo gasto no envio de uma mensagem pode ser razoavelmente bem modelado pela equação:

$$T_{msg} = t_s + t_w L$$

onde t_s é o tempo de inicialização da mensagem (startup time), é o tempo de transferência por palavra, e L é o comprimento da mensagem em palavras.

Largura de banda (bandwidth) do canal de comunicação é

Tipicamente, é aprox. duas ordens de grandeza maior do que

Startup dominant para msg pequena, BW domina para msg grande.

Roteamento de comunicação

Modelos mais sofisticados de tempo de comunicação podem considerar distância entre processadores ou contenda para BW entre processadores

P.ex. Roteamento "store-and-forward" pode ser modelado como

$$T_{msg} = (t_s + t_w L) D$$

onde D é a distância em 'pulos' entre proc. que envia e proc. que recebe

Processadores modernos usam roteamento 'cut-through' ou 'worm-hole', que pode ser modelado por:

$$T_{msg} = t_s + t_w L + t_h D$$

onde t_h é o custo incremental por pulo para mandar a mensagem. Tipicamente é desprezível.

Contenda para comunicação

Contenda para BW pode ser modelada por

$$T_{msg} = t_s + t_w SL,$$

onde S é o número de processadores precisando enviar mensagens 'pelo mesmo fio' simultaneamente.

Cada processador fica efetivamente com $1/S$ do BW disponível.

Tempo ocioso

Tempo ocioso se deve a falta de tarefa alocada ou falta de dados necessários (p.ex. na espera da chegada de uma mensagem).

Tempo ocioso decorrente de falta de tarefa pode ser reduzido pela melhoria no balanceamento de carga.

Tempo ocioso decorrente da falta de dados pode ser reduzido pela utilização de computação e comunicação sobrepostas ou assincronismo.

Multithreading é um enfoque para sobrepor comunicação e computação.

Exemplo

Referências

G. M. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities, *Proc. AFIPS*, 30:483-485, 1967.

J. L. Gustafson, "Reevaluating Amdahl's law," *Comm. ACM* 33:539-543, 1990.

J. P. Singh, J. L. Henessy & A. Gupta, "Scaling parallel programs for multiprocessors: methodology and examples," *IEEE Computer*, 26(7):42-50, 1993.

X. H. Sun & L. M. Ni, "Scalable problems and memory-bound speedup," *J. Parallel Distrib. Comput.*, 19:27-37, 1993.

A. Grama, A. Gupta & V. Kumar, "Isoefficiency: measuring the scalability of parallel algorithms and architectures," *IEEE Parallel Distrib. Tech.* 1(3):12-21, 1993.

P. H. Worley, "The effect of time constraints on scaled speedup," *SIAM J. Sci. Stat. Comput.*, 11:838-858, 1990.

Redes de interconexão

Redes de interconexão

Acesso a dados remotos em computador paralelo requer comunicação entre processadores (ou entre processadores e memória).

Conexão direta ponto-a-ponto entre um número elevado de processadores (ou memórias) é inviável porque exigiria $\boldsymbol{O}(p^2)$ 'fios'.

Conexões diretas são feitas apenas entre alguns pares de processadores (ou memórias), de modo que roteamento entre processadores intermediários ou chaves é necessário para comunicação entre pares não conectados.

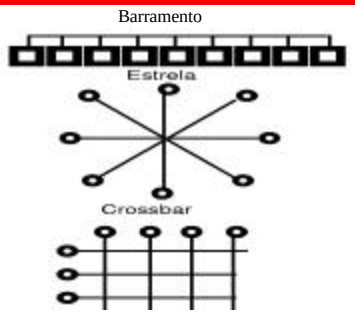
Topologia da rede resultante, esparsamente conectado, determina parcialmente a latência e largura de banda (BW) comunicante.

Topologias de redes

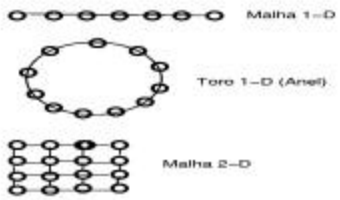
Muitas topologias tem sido propostas ou construídas, porém a maioria dos computadores comercialmente disponíveis são baseados em uma das seguintes topologias:

- Barramento
- Crossbar
- Malha ou toro 1-D, 2-D, ou 3-D
- Hipercubo
- Árvore
- Butterfly (borboleta)

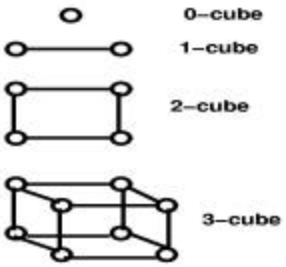
Redes baseadas em barramentos e crossbar



Redes em Malha e Toro

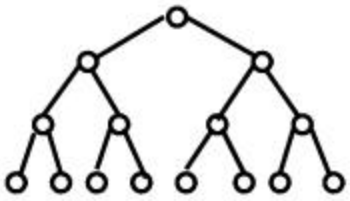


REDES HIPERCÚBICAS



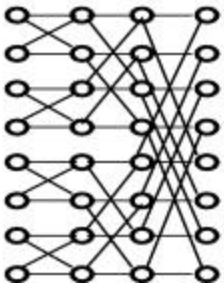
REDES EM ÁRVORE

árvore binária



REDES MULTI ETAPA

butterfly



Propriedades de topologias de rede

- *Grau*: número máximo de arestas emanando de qualquer nó.
- Diámetro: distância máxima (número de pulos) entre qq. par de nós
- Largura de biseção: menor número de arestas cuja retirada decompõe rede em duas 'metades' iguais.
- Comprimento físico máximo de qualquer aresta.

Propriedades de topologias de rede

Rede	Nós	Grau	Diam	BW	Comprim.
Bus/estrela	k	k	2	1	var
Malha dim d	k^d	2d	d (k – 1)	k ^(d-1)	Cte.
Árvore binar.	2^k – 1	3	2(k-1)	1	Var
Hipercubo	2^k	k	k	2^(k-1)	var
Butterfly	(k+1)2^k	4	2k	2^k	var

Topologias Práticas de Redes

A maioria de SMPs utiliza barramento ou rede crossbar e fica limitado a um número reduzido de processadores.

Para MPPs, redes hipercúbicas eram adotadas incialmente, principalmente em função da sua elegância algorítmica, flexibilidade, diâmetro baixo e alta BW.

Porém, o grau e comprimento de aresta variáveis complicam projeto e fabricação de redes hipercúbicas.

A maioria de MPPs contemporâneos usem malhas 2-D ou 3-D que possuem graus e comprimentos de arestas constantes o que se adequa ao caso de algoritmos baseados em malhas (grid-based algorithms)

Máquinas DSM geralmente são híbridas, possuindo conectividade completa (ou barramento) nos clusters locais, e menos conexões entre clusters.

Mapeando grafo de tarefas à topologia da rede

Mapear o grafo de tarefas de um determinado problema à topologia da rede da máquina alvo pode ser visto como um problema de imersão em grafos.

Para o mapeamento $f : G_1(V_1, E) \rightarrow G_2(V_2, E_2)$,

- *Dilatação* é a distância máxima entre quaisquer dois nós $f(x)$ $f(y)$ em G_2 tais que x e y sejam adjacentes em G_1.
- *Carga* é o número máximo de nós em V_1 mapeado em um único nó em V_2.
- *Congestão* é o número máximo de arestas em E_1 mapeado em única aresta em E_2.

Mapeando grafo de tarefas à topologia da rede

Idealmente, queremos dilatação, carga e congestão igual a 1, porém nem sempre é possível.

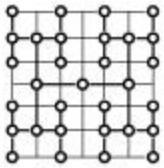
Por exemplo, um anel possui imersão perfeita em malha 2-D com o mesmo número de nós se e somente se a malha possuir um número par de linha e/ou colunas.

Mapeando grafo de tarefas à topologia da rede

Determinar a melhor imersão possível entre dois grafos é um problema combinatórico difícil (NP-completo), de modo que geralmente utiliza-se uma heurística.

Por outro lado, vários casos particulares que surgem em aplicações práticas possuem soluções boas, ou até ideais, conhecidas.

EXEMPLO DE IMERSÃO



árvore com k níveis possui imersão em malha 2-D

Dilatação: ceil([k-1)/2]

Imersão de árvore binária em malha 2-D

Imersões em hipercubos

Um aspecto atraente de topologia hipercúbica é que a imersão de vários outros tipos de grafos pode ser feita nela, freq. ideal.

Por exemplo, uma imersão perfeita de uma malha 2 -D ou toro com 2^j X 2^k processadores pode ser feita em hipercubo com 2^(j+k) processadores.

Esta imersão pode ser realizada utilizando o código de Gray, no qual inteiros de 0 a 2^n – 1são ordenados tal que as representações binárias de membros consecutivos da sequência diferem em exatamente uma posição (bit).

Código de Gray Refletido

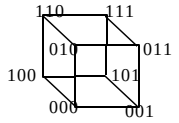
Por exemplo, o código binário de Gray refletido de comprimento 16 é:

0000	0
0001	1
0011	3
0010	2
0110	6
0111	7
0101	5
0100	4

1100	12
1101	13
1111	15
1110	14
1010	10
1011	11
1001	9
1000	8

Imersão de anel em hipercubo

Dois nós de um hipercubo são conectados sss seus números diferem em exatamente uma posição de bit



000 ¸001 ¸ 011 ¸010 ¸110 ¸ 111 ¸101 ¸ 100 ¸000

Malha ou toro de dimensão maior pode ser ‘mergulhado’ em hipercubo de dimensão adequada pela concatenação de códigos de Gray para cada numeração de coordenadas.

Paradigma de hipercubo

Muito embora redes hipercúbicas são infrequentes em máquinas comerciais hoje em dia, ainda servem como um paradigma importante para muitos algoritmos paralelos, como, por exemplo, operações de comunicação coletiva.

Referências

•L. N. Bhuyan, Q. Yang & D. P. Agarwal, "Performance of multiprocessor interconnection networks", *IEEE Computer* 22(2):25-37, 1989.

•O. Krämer & H. Mühlenbein, "Mapping strategies in message-based multiprocessor systems," *Parallel Computing*, 9:213-225, 1989.

•V. Lo, "Heuristic algorithms for task assignment in distributed systems," *IEEE Trans. Comput.*, 37:1384-1397,1988.

•Y. Saad & M. H. Shultz, "Topological properties of hypercubes," *IEEE Trans. Comput.*, 37:867-872, 1988.

Comunicação entre processadores

Roteamento de mensagens

Se a mensagem for mandada de um processador a outro não ligado (físicamente) a ele, então ela tem que ser roteada por meio de outros que estão conectados entre si.

Algoritmos para roteamento podem ser

- Minimal ou não minimal
- Estáticos ou dinâmicos
- Determinístico ou randômizados

A maioria de topologias regulares admite esquemas de roteamento relativamente simples do tipo estático, determinístico, e minimal

Roteamento de mensagens

Em uma malha 2-D ou toro, por exemplo, a mensagem pode ser transmitido adiante ao longo de uma linha até chegar na coluna do processador destino, e daí transmitido adiante ao longo daquela coluna até chegar no processador destino (ou na ordem inversa).

No hipercubo, se número do nó atual difere do número do nó destino no i-ésimo bit, então a mensagem é passada adiante ao processador adjacente que tenha valor oposto naquele bit.

Desta forma, a mensagem chegará no destino em k passos, onde k é o número de posições onde os números dos nós fonte e destino diferem em bits. Claramente k não pode exceder a dimensão do hipercubo.

Roteamento de mensagens

Há liberdade considerável na escolha de um esquema de roteamento.

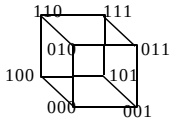
Por exemplo, numa malha 2-D ou 3-D, podemos considerar as dimensões respectivas em qualquer ordem.

Em um hipercubo, bits que diferem entre nó fonte e nó destino podem ser 'corrigidos' em qualquer ordem.

Portanto, frequentemente existem caminhos múltiplos possíveis para uma dada mensagem, e esta liberdade pode ser explorada para melhorar desempenho e/ou tolerância a falhas.

Imersão de anel em hipercubo

Dois nós de um hipercubo são conectados se e somente se seus números diferem em exatamente uma posição de bit

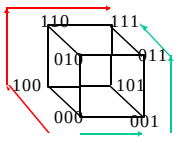


000 , 001 , 011 , 010 , 110 , 111 , 101 , 100 , 000

Malha ou toro de dimensão maior pode ser 'imergido' em hipercubo de dimensão adequada pela concatenação de códigos de Gray para cada numeração de coordenadas.

Roteamento de mensagens em hipercubo

Exemplo: Nós 0 (000) e 7 (111) não são fisicamente ligados, portanto uma mensagem de nó 0 a nó 7 pode ser roteada assim: 0 , 1 , 3 , 7



Mostrando duas rotas diferentes de nó 0 (fonte) até nó 7 (destino)

Bits no endereço do nó podem ser considerados em qq. Ordem, esq. à dir. (como no exemplo acima) ou vice-versa. Cada escolha gera um caminho distinto de fonte até destino.

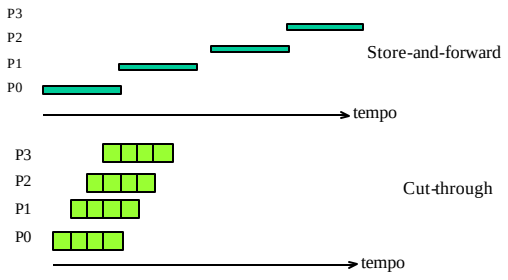
Roteamento 'cut-through'

Os primeiros computadores de memória distribuida utilizavam roteamento *store-and-forward* : em cada processador ao longo do caminho da fonte até o destino, a mensagem inteira é recebida e armazenada antes de ser despachada adiante.

Um desempenho superior é atingida na maioria das redes modernas de comunicação utilizando roteamento *cut-through* (ou *wormhole*), no qual a mensagem é quebrada em segmentos menores que são transmitidos através da rede using pipeline.

Cada processador no caminho passa adiante cada segmento assim que recebe, proporcionando um aumento na velocidade de comunicação bem como permitindo uma redução na capacidade do buffer.

Store-and-forward vs cut-through



Roteamento cut-through

Roteamento cut-through estabelece circuito virtual entre processadores fonte e destino.

É preciso tomar cuidado no algoritmo de roteamento para evitar impasses (deadlock) eventuais, quando múltiplas mensagens precisam utilizar o mesmo link no mesmo instante.

Este roteamento faz com que o diâmetro da rede seja um parâmetro menos crucial para mensagens individuais, de modo que usuários podem se preocupar menos com o casamento entre a topologia do problema e da rede.

Porém, restrições sobre o BW total ainda podem exigir alguns cuidados com localidade no projeto de algoritmos paralelos.

Concorrência de comunicação

Até agora consideramos apenas comunicação ponto-a-ponto entre um par de processadores.

Quando múltiplos processadores se comunicam simultaneamente, o desempenho global atingível é afetado pelo grau de concorrência suportado pelo mecanismo de comunicação subjacente.

Em uma determinada arquitetura pode ser possível (ou não):

•Mandar e receber pelo mesmo link simultaneamente

•Mandar por um link e receber por outro simultaneamente

•Mandar e/ou receber por múltiplos links simultaneamente.

Concorrência de comunicação

Podemos englobar todas estas variações através de uma definição apropriada de um ‘passo’de uma determinada padrão de comunicação

O resultado é a multiplicação do custo total por um fator constante em uma rede cujo grau não varia em função do número de processadores (e.g., uma malha).

Por outro lado, este fator correspondente pode crescer em função do número de processadores em uma rede de grau variável como a rede hipercúbica.

Comunicação coletiva

Comunicação coletiva envolve múltiplos processadores simultaneamente. Os exemplos mais frequentes na prática são:

- Broadcast: um-a-todos
- Redução: todos-a-um
- Broadcast multi-nó: todos-a-todos
- Scatter/gather: um-a-todos/todos-a-um
- Intercâmbio total: todos-a-todos personalizado
- Shift (deslocamento) circular
- Barreira

A melhor implementação depende da topologia e do esquema de roteamento.

Broadcast

No *broadcast*, um processador comunica uma mensagem a $p - 1$ outros processadores.

Processador fonte poderia mandar, sequencialmente, $p - 1$ mensagens separadamente, uma a cada processador.

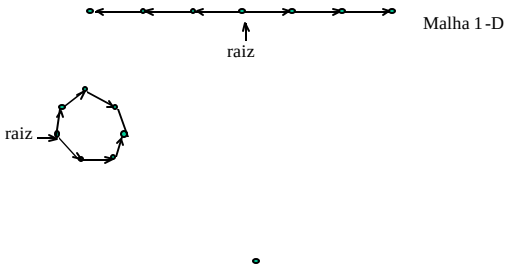
Porém, eficiência pode ser melhorada explorando paralelismo e o fato de ter que usar processadores intermediários.

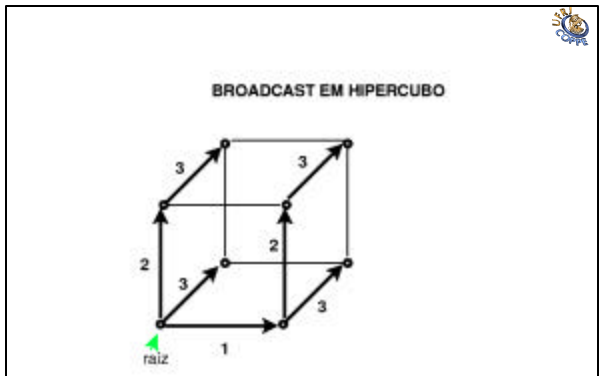
Algoritmo de broadcast

Algoritmo genérico de broadcast:

1. Se fonte ® eu, receba mensagem.
2. Mande mensagem a cada um dos vizinhos que não a tenham recebido.

Broadcast em malha ou toro





Custo de broadcast

Algoritmo de broadcast cria árvore geradora para uma determinada topologia de rede, com processador fonte como raiz da árvore.

Altura da árvore geradora determina o número total de passos exigidos. Desta forma, o custo total para uma mensagem de comprimento L é:

- Malha 1-D: $(p-1)(t_s + t_w L)$
- Malha 2-D: $2(\sqrt{p}-1)(t_s + t_w L)$
- Hipercubo: $\log(p)(t_s + t_w L)$

Broadcast incrementado

Para mensagens longas para as quais bandwidth domina latência, BW da rede pode ser melhor explorada quebrando a mensagem em pedaços e

- Mandar pedaços em modo pipeline usando uma única árvore geradora
- OU mandar cada pedaço utilizando diferentes árvores geradoras (com a mesma raiz)

Em um hipercubo com 2^*k* nós, por exemplo, dado qq. nó como raiz, existem *k* árvores geradoras (com arestas disjuntas), que podem ser utilizadas (potencialmente) simultaneamente em um broadcast.

Redução

Em ***redução***, dados de todos os processadores são combinados utilizando uma dada operação associativa (e.g., soma, produto, max, min, OU lógico, E lógico) para produzir o resultado final.

Como em broadcast, estrutura de comunicação utiliza uma árvore geradora para uma dada rede, porém fluxo de dados é na direção oposta, das folhas à raiz.

Resultados (intermediários) entrando são combinados com os valores do processador que recebe antes de enviar para o pai.

Resultado final acaba acumulando no processador raiz. Se os outros processadores também precisam dele, pode-se recorrer a um broadcast usual.

Algoritmo de redução

Algoritmo genérico de redução:

1. Receba mensagem de todos os meus filhos na árvore geradora (se há).
2. Combina valores recebidos com o meu utilizando a operação associativa especificada.
3. Mande resultado ao meu pai, se há.

Os desenhos de algoritmos de redução para as topologias diferentes são os mesmos do broadcast, com a direção das setas invertidas.

Redução em malha ou toro

Redução em hipercubo

Custo de redução

Algoritmo de redução utiliza a mesma árvore geradora que o broadcast, porém no sentido invertido.

Altura da árvore geradora determina o número total de passos exigidos. Desta forma, o custo total para uma mensagem de comprimento L é:

•Malha 1-D:

•Malha 2-D: $\sqrt{}$

•Hipercubo:

onde t_c é o custo por palavra da operação associativa de redução.

Broadcast multi nó

Em broadcast multi nó, cada processador manda a sua mensagem a todos os demais.

Esta operação todos-a-todos é logicamente equivalente a *p* broadcasts um-a-todos, e poderia ser implementado assim.

A eficiência poderia, porém, ser melhorada pela sobreposição de vários broadcasts separados.

Em um anel, o primeiro passo de um broadcast unidirecional pode ser iniciado a partir de cada nó no mesmo instante.

Depois de *p*-1 passos, cada processador terá recebido dados de todos os demais, completando a operação todos-a-todos.

Broadcast multi nó

Em um toro 2-D, algoritmo do anel pode ser aplicado primeiro em cada linha, e depois em cada coluna (ou vice-versa).

Em um hipercubo com 2^k processadores, broadcast multi nó pode ser implementada por sucessivas trocas em pares em cada uma das *k* dimensões, com mensagens concatenadas em cada etapa.

Redução via Broadcast multi nó

Se, ao invés de concatenar as mensagens, elas forem combinadas utilizando uma operação associativa apropriada, broadcast multi nó pode ser utilizada para implementar redução.

Este enfoque possui a vantagem de evitar o broadcast do resultado final (após redução ao único nó raiz), portanto representa uma diminuição (até pela metade) do custo.

Comunicação coletiva personalizada

Em um broadcast, nó(s) raiz enviam a *mesma* mensagem aos outros.

Nas versões análogas personalizadas, mensagens *distintas* são enviadas aos outros processadores.

Scatter (espalhar) é CCP análogo ao broadcast, exceto que raiz manda mensagens *diferentes* para cada processador.

Gather (juntar) é CCP análogo a redução, exceto que os dados recebidos pelo nó raiz são concatenados ao invés de reduzidos através de uma operação associativa.

Intercâmbio total é broadcast multi nó personalizada todos-a-todos.

Comunicação coletiva personalizada

Operações CCP são implementadas por algoritmos parecidos com aqueles já vistos.

Scatter, por exemplo, utiliza a mesma árvore geradora que o broadcast padrão, porém múltiplas mensagens são transmitidas juntas em cada etapa.

Nó raiz envia mensagem a cada filho contendo dados para a sub-árvore inteira da qual aquele filho é raiz. Cada filho extrai seus dados e despacha o resto a cada um de seus filhos da mesma maneira, até que cada nó recebe sua mensagem.

Prefixo ou scan

Deslocamento circular

Em k-deslocamento circular (circular k-shift), com $0 < k < p$, processador i envia dados a processador $(i + k) \bmod p$.

Tais operações ocorrem em problemas de computação matricial, diferenças finitas e casamento de strings (string matching).

Em uma rede anelar, há implementação natural para k-deslocamento circular .

Implementação de k-deslocamento circular em outras topologias de rede pode ser complicada, porém basicamente envolve a imersão de um anel (ou série de anéis) na rede em questão.

Barreira

Uma ***barreira*** é um mecanismo de sincronização: todos os processadores têm que chegar nela antes que qualquer um passe adiante.

Implementação de uma barreira depende da arquitetura da memória e rede subjacentes.

Em sistemas de memória distribuida, a barreira é usualmente implementada por troca de mensagens, utilizando algoritmos parecidos com aqueles de comunicação todos-a-todos.

Em sistemas de memória compartilhada, barreira é usualmente implementada utilizando semáforos, test-and-set ou outros mecanismos para impor exclusão mútua.

Referências

M. Barnett, R. Littlefield, D. Payne & R. van de Geijn, 'Global combine algorithms for 2-D meshes with wormhole routing', *J. Parallel Distrib. Computing*. 24:191-201,1995.

M. Barnett, D. Payne, R. van de Geijn & J. Watts 'Broadcasting on meshes with wormhole routing', *J. Parallel Distrib. Computing*. 35:111-122,1996.

D. P. Bertsekas, C. Özveren, G.D. Stamoulis, P. Tseng & J.N. Tsitsiklis, "Optimal communication algorithms for hypercubes", *J. Parallel Distrib. Computing* 11:263-275,1991.

P. Kermani & L. Kleinrock, "Virtual cut-through: a new communication switching technique", *Computer Networks* 3:267-286, 1979.

L. M. Ni & P. McKinley, "A survey of wormhole routing techniques in direct networks," *IEEE Computer* 26(2):62-76, 1993.

Y. Saad & M. Shultz, "Data communication in parallel architectures," *Parallel Computing* 11:131-150, 1989

R. Van de Geijn, "On global combine operations", *J. Parallel Distrib. Comput.* 22: 324-328, 1994

MPI: Message-Passing Interface

MPI

MPI é padrão para escrever programas paralelos utilizando troca de mensagens.

MPI não é uma linguagem e sim uma biblioteca de rotinas que podem ser chamadas a partir de linguagens convencionais como FORTRAN, C ou C++.

MPI fornece comunicação entre múltiplos processos concorrentes, cada um dos quais executa um programa sequencial, que podem ou não ser idênticos.

MPI se aproxima bem a nossa metodologia para desenvolver algoritmos paralelos e fornece um mecanismo natural para a sua implementação.

MPI roda em quase qq arquitetura e plataforma paralela.

MPI

MPI é mais portátil que outros pardigmas para escrever programas paralelos, e pelo fato de abilitar e estimula atenção à localidade de dados, MPI frequentemente possui desempenho melhor que os demais.

MPI é grande e complexo, com mais de 125 funções e muitas opções e protocolos diferentes disponíveis.

Para a maioria das aplicações, porém, um pequeno subconjunto basta.

Falaremos apenas dos aspectos mais essenciais de MPI.

MPI-1

MPI-1 inclui

- Comunicação ponto-a-ponto
- Operações de comunicação coletiva
- Grupos de processos e domínios de comunicação
- Topologias virtuais de processos
- Gerenciamento de ambiente e pesquisa
- Interface de perfilamento
- Bindings para FORTRAN e C

Falaremos apenas da parte de MPI que basta para implementar (não necessariamente da maneira mais eficiente) dos algoritmos que serão discutidos no âmbito deste minicurso.

MPI-2

MPI-2 inclui

- Gerenciamento dinâmico de processos
- Entrada/saída
- Operações de comunicação unilaterais
- Bindings para C++

Não falaremos de MPI-2, que não é essencial para os algoritmos considerados neste curso e ainda não está universalmente disponível em toda plataforma paralela.

Linguagens de programação

MPI inclui bindings para FORTRAN, C, C++

Daremos alguns exemplos de bindings para C; FORTRAN é parecido

Uma diferença significativa é que versões em C de várias rotinas MPI devolvem um código de erro como valor da função, ao passo que as versões em FORTRAN possuem um argumento inteiro adicional, `IERROR`, para este fim.

Grupos e comunicadores

Cada processo MPI pertence a um ou mais ***grupos***.

Cada processo é identificado pelo seu ***rank*** dentro de um dado grupo, onde rank é um inteiro de zero até um menos do tamanho do grupo.

Inicialmente, todo processo pertence a `MPI_COMM_WORLD`, no qual cada possui rank entre zero e um a menos do que o número total de processos.

Grupos adicionais podem ser criados pelo usuário.

Visto como um domínio de comunicação ou contexto, um grupo de processos é chamado de ***comunicador*** em MPI

Especificação e indentificação de mensagens

Em qq sistema de comunicação, vários informações são necessárias para especificar uma mensagem e identificar sua fonte e seu destino

Em MPI esta informação inclui:

•Address – local na memória onde dados da mensagem começam

•Count – número de ítens de dados contido na mensagem

•Datatype – tipo de dado na mensagem

•Source ou destination – rank de processo no comunicador que envia ou recebe

•Tag – identificador para mensagem específica ou tipo de msg

•Communicator – domínio de comunicação ou contexto

Iniciação e término de MPI

Inicialize MPI:

```
Int MPI_init(int *argc, char ***argv)
```

Nenhuma função de MPI pode ser chamada antes de `MPI_init`

Término de MPI

```
Int MPI_Finalize(void)
```

Nenhuma função MPI pode ser chamada depois de `MPI_Finalize`

Pesquisa do ambiente

Determine número de processadores:

```
Int MPI_Comm_size(MPI_Comm comm, int *size)
```

Retorna na variável `size` o número de processos no grupo `comm`.

Determine identificador de processo atual:

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

No retorno, `rank` contém rank do processo atual no grupo `comm` ,

$0 \leq$ `rank` $\leq$ `size - 1.`

Enviando e Recebendo Mensagens

Enviar mensagem:

```
int  MPI_Send(void *buf, int count, MPI_Datatype
datatype, int dest int tag, MPI_Comm comm)
```

Receber mensagem:

```
int  MPI_Recv(void *buf, int count, MPI_Datatype
datatype, int source int tag, MPI_Comm comm,
MPI_Status *status)
```

Enviando e recebendo mensagens

Tipos de dados disponíveis incluem os familiares como `char, int, float, double` (em C) e `INTEGER, REAL, DOUBLE PRECISION` (em FORTRAN)

Uso dos tipos de dados de MPI facilita ambiente heterogêneo no qual tipos nativos podem variar de máquina em máquina.

MPI também suporta tipos de dados definidos por usuário par dados contíguos ou não contíguos.

Valores wildcard `MPI_ANY_SOURCE` e `MPI_ANY_TAG` podem ser usados para source e tag, para receber mensagem.

Valores de source e tag podem então ser determinados a partir dos campos `MPI_SOURCE` e `MPI_TAG` da estrutura `status`

Enviando e recebendo mensagens

Ambas as funções padrão send e receive são ***bloqueadoras*** no sentido de recursos especificados na chamada, tais como o buffer da mensagem, podem ser reutilizados com segurança imediatemente após retorno

Em particular, `MPI_Recv` retorna somente após o buffer conter a mensagem requerida.

`MPI_Send` pode ser iniciado antes ou depois `MPI_Recv` correspondente é iniciado.

Dependendo da implementação do MPI, `MPI_Send` pode retornar antes ou depois o início do `MPI_Recv` correspondente (casado).

Para o mesmo `source, tag, e comm,` mensagens são recebidas no destino na ordem em que foram mandadas.

MPI Básico

Seis funções MPI descritas até agora bastam para implementar quase qq algoritmo paralelo de maneira razoavelmente eficiente.

Outras 120 (aprox) funções MPI fornecem conveniência, flexibilidade, robustez, modularidade, e eficiência potencialmente maiores.

Porém também introduzem complexidade adicional considerável que pode ser de difícil gerenciamento.

Por exemplo, algumas destas funções facilitam sobreposição de comunicação e computação, mas encarregam o usuário com a tarefa de sincronização

Programa exemplo utilizando MPI

```
#include <mpi.h>
void main(int argc, char **argv) {
int p, i, left, right, count, tag, nit,k;
float y, yl, yr;
MPI_Status status;
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &i);
count = 1; tag = 1; nit = 10;
left = i-1; right = i+1;
If (i==0) y = alpha;
  else if (i== p-1) y =beta
  else y = 1.0;
```

Programa exemplo utilizando MPI

```
for (k=1; k<= nit; k++) {
  if (i != 0)
     MPI_Send(&y, count, MPI_FLOAT, left, tag, MPI_COMM_WORLD);
  if (i != p-1) {
     MPI_Recv(&yr, count, MPI_FLOAT, right, tag,
MPI_COMM_WORLD, status);
     MPI_Send(&y, count, MPI_FLOAT, right, tag,
MPI_COMM_WORLD);
   }
  if (i != 0)
     MPI_Recv(&yl, count, MPI_FLOAT, left, tag, MPI_COMM_WORLD,
&status);
  y = (yl+ yr)/2.0;
  }
}
```


Criando e executando Programas MPI

Para rodar um programa MPI, módulo executável deve ser criado primeiro, compilando programa do usuário e linkando com a biblioteca MPI.

Pode ser preciso utilizar um ou mais arquivos de cabeçalho como `mpi.h` para fornecer as definições e declarações necessárias.

MPI é comumente utilizado no modo SPMD, de modo que apenas um executável tem que ser criado, e daí múltiplas instâncias dele são executadas concorrentemente.

Comando para iniciação tipicamente chamado `mpirun`, cujas opções incluem número de processos a serem criados, possivelmente os processadores nas quais estes processos devem rodar, etc.

Medindo tempo de execução

`MPI-Wtime()` retorna tempo wall-clock (elapsed) em segundos a partir de algum instante arbitrário do passado. O valor retornado é um número ponto flutuante, precisão dupla.

Tempo (elapsed time) para um dado segmento de programa pode ser medido chamando `MPI-Wtime()` no início e no final e calculando a diferença.

Valores de relógio para processos diferentes não são necessariamente comparáveis, pois não são necessariamente sincronizados.

Mesmo que sejam sincronizados, variação entre eles não é significativamente menor que o tempo de ida e volta de uma mensagem de comprimento zero entre dois processos.

Modos de comunicação

O modo padrão de MPI não especifica se mensagens são colocados em buffer, porém deixa esta decisão a critério da implementação.

Desta forma, no modo padrão, portabilidade exige hipóteses conservadores em relação a ordem de início de send e recv a fim de evitar impasses (deadlock) potenciais.

MPI fornece três modos adicionais de comunicação que permite usuários controlar se mensagens são colocados em buffers ou não, para que a ordenação correta de sends e receives não seja ao acaso.

Modos de comunicação

No modo ***buffered***, send pode ser iniciado independentemente do início da recepção do receive casado, e send pode ser terminado antes do início do receive casado.

No modo ***síncrono***, send pode ser iniciado independente do início do receive casado , porém send terminará somente após início do receive casado.

No modo ***ready***, send pode ser iniciado somente se receive casado já foi iniciado.

As funções MPI correspondentes são, respetivamente, `MPI_Bsend`, `MPI_Ssend`, e `MPI_Rsend` . O mesmo `MPI_Recv` funciona para todos os modos de send.

Modo buffered requer alocação de espaço utilizando `MPI_Buffer_attach`

Comunicação não bloqueante

Todas as funções de comunicação discutidas até agora são bloqueantes no seguinte sentido: recursos especificados na chamada, como o buffer da mensagem, podem ser reutilizados (com segurança) imediatamente após retorno.

MPI também oferece funções não bloqueantes para cada modo de send: `MPI_Isend`, `MPI_Ibsend`, `MPI_Irsend`, e `MPI_Issend`.

Única função receive não bloqueante é `MPI_Irecv`.

Funções não bloqueantes incluem argumento adicional `request` que pode ser testado para verificar se operação solicitada já terminou.

Funções `MPI_Test` e `MPI_Wait` são usadas para testar ou esperar, respetivamente, o término de operações não bloqueantes.

Comunicação persistente

MPI oferece opções para otimizar operações de comunicação que são executadas repetidas vezes com a mesma lista de argumentos.

Operações persistentes de comunicação 'amarram' lista de argumentos à solicitação. Daí solicitação pode ser utilizada repetidas vezes para iniciar e completar mensagens sem repetir a lista de argumentos cada vez.

Uma vez que a lista de argumentos foi amarrada utilizando `MPI_Send_init` ou `MPI_Recv_init`, por exemplo, (ou de maneira similar para outros modos), então solicitação pode ser iniciada repetidas vezes usando `MPI_Start`.

Pesquisando e cancelando mensagens

Às vezes é útil pesquisar mensagens sem efetivamente recebê-las.

Por exemplo, informação sobre uma mensagem determinada a partir do valor do argumento `status` retornado por uma pesquisa pode ser utilizada para determinar como recebê-la.

Funções `MPI_Pprobe` e `MPI_Iprobe`, que são bloqueantes e não bloqueantes, respetivamente, fornecem esta capacidade.

Também é possível cancelar uma solicitação pendente de uma mensagem utilizando `MPI_Cancel`, que é útil na fase de limpeza.

Comunicação coletiva

Funções de comunicação coletiva fornecidas pelo MPI incluem;

- `MPI_Bcast`
- `MPI_Reduce`
- `MPI_Allreduce`
- `MPI_Alltoall`
- `MPI_Scatter`
- `MPI_Gather`
- `MPI_Scan`
- `MPI_Barrier`

Manipulação de comunicadores

Funções fornecidas pelo MPI para a manipulação de comunicadores incluem:

- `MPI_Comm_create`
- `MPI_Comm_dup`
- `MPI_Comm_split`
- `MPI_Comm_compare`
- `MPI_Comm_free`

Topologias virtuais de processos

MPI fornece topologias virtuais de processos correspondentes a grades regulares cartesianos, bem como grafos mais gerais.

A topologia é um atributo opcional acicional que pode ser dado ao comunicador.

A definição de uma topologia virtual de processos pode

•facilitar a aplicação. Por exemplo, grade cartesiana 2 -D é natural para muitas operações de álgebra matricial.

•ajudar MPI a alcançar um mapeamento mais eficiente dos processos à uma determinada rede física.

• `MPI_Topo_test` determina o tipo de topologia definida (se há) para um dado comunicador.

Topologia de Grade Cartesiano

`MPI_Cart_create` cria a topologia cartesiana. O usuário pode especificar dimensão, número de processadores por dimensão, e se a dimensão é periódica (i.e., possui wrap-around)

Hipercubos são contemplados, uma vez que hipercubo de dimensão *k* e toro de dimensão *k* com dois processos por dimensão.

Funções de pesquisa obtém informações sobre topologia cartesiana existente, como dimensão, número de processadores por dimensão, e se a dimensão é periódica.

Também existem funções para determinar coordenadas de um process o.

`MPI_Cart_shift` fornece deslocamento fixo ao longo de uma dimensão.

Topologia de grafo geral

`MPI_Graph_create` cria topologia de grafo geral. O usuário pode especificar número de nós, grau de cada nó, e as arestas do grafo.

Funções de pesquisa obtém informações sobre topologia existente, como número de nós e arestas, listas de graus e arestas, númerode vizinhos de um dado nó, lista de vizinhos de um dado nó.

Acesso ou download de MPI

Versões customizadas de MPI são fornecidas por quase todos os fabricantes de computadores paralelos atuais.

Adicionalmente, versões freeware de MPI são disponíveis para redes de estações e ambientes parecidos (cluster de PCs) nos sites:

http://www.mcs.anl.gov/mpi (Argonne National Laboratory, EUA)

http://www.mpi.nd.edu/lam (Univ de Notre Dame (orig Ohio SC)

Ambos os sites também disponibilizam tutoriais e material didático sobre MPI.

MPI: referências

W. Gropp, E. Lusk & A. Skjellum, *Using MPI: Portable parallel programming with the message-passing interface*, MIT Press, 1994.

P. S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, 1997.