

Programação Paralela na Plataforma Microsoft

Alexandre Ricardo Nardi (Microsoft)
Nicolas Maillard (UFRGS)

Computação de Alto Desempenho

- **Quando usar?** Problemas "difíceis" necessitam de alto desempenho (ciência, engenharia e negócios)
- **Por que usar?** Pressões do mercado pedem ciclo de inovação acelerado e redução de custo
 - Design de aeronaves utilizando materiais compostos
 - Melhoria na eficiência e segurança dos combustíveis de veículos
 - Simulação da catálise de enzimas e proteínas
 - Design de medicamentos
 - Simulação de dispositivos eletrônicos em escala nano
 - Modelagem de risco em portfólios financeiros
 - Criação e aprimoramento de conteúdo digital
 - Modelagem e otimização da cadeia de suprimentos
 - Previsão do clima



2

A Race for Flops...

Since 2000, supercomputers have left the U.S. in the dust

Rank	System	Location
1	IBM SP6 Supercomputer	Los Alamos
2	IBM SP6 Supercomputer	Los Alamos
3	IBM SP6 Supercomputer	Los Alamos
4	IBM SP6 Supercomputer	Los Alamos
5	IBM SP6 Supercomputer	Los Alamos
6	IBM SP6 Supercomputer	Los Alamos
7	IBM SP6 Supercomputer	Los Alamos
8	IBM SP6 Supercomputer	Los Alamos
9	IBM SP6 Supercomputer	Los Alamos
10	IBM SP6 Supercomputer	Los Alamos

...And Tera is Just the Beginning

Power needed to complete most complex task in one week

Task	Peak Speed
Automotive Development	100 TeraFLOPS
Human Vision Simulation	100 TeraFLOPS
Aerodynamic Analysis	1 PetaFLOPS
Laser Optics	10 PetaFLOPS
Molecular Dynamics in Biology	20 PetaFLOPS
Aerodynamic Design	1 ExaFLOPS
Computational Cosmology	10 ExaFLOPS
Turbulence in Physics	100 ExaFLOPS
Computational Chemistry	1 ZettaFLOP

FLOP: float point operations per second

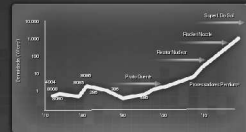
3

Supercomputadores Brasileiros?

	1991	1998	2005
Sistema	Cray Y-MP C916	Sun HPC10000	Shuttle @ NewEon.com
Arquitetura	16 x Vector 4GB, Bus	24 x 333MHz UltraSPARCII, 24GB, SBus	4 x 2.2GHz x64 4GB, GIG
SO	UNICOS	Solaris 2.5.1	Windows Server 2003
Performance	~10 GFlops	~10 GFlops	~10 GFlops
Top500 #	1	500	N/A
Preço	\$40,000,000	\$1,000,000 (40x)	< \$4,000 (250x drop)
Clientes	Lab. Governo	Grandes Empresas	Engenheiros e
Aplicações	Secretas, Clima, Pesquisa em	Manufatura, Energia, Finanças, Telecom	Biotecnologia, Ciências, Mídia Digital

4

Bem Vindo à era Multi-Core



5

Alternativas de Paralelização

OpenMP
Message Passing Interface
Parallel Computing Platform

6

OpenMP

7

Open MP

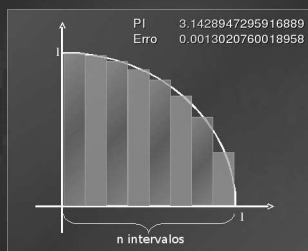
- p/ Multi-CPU c/ Memória Compartilhada
- Baseada em pragmas sobre C/C++ (desde VS2005)

```
#pragma omp parallel for
for (i=1; i<n; i++)
    b[i] = (a[i] + a[i-1]) / 2.0;

#pragma omp parallel sections
{
    #pragma omp
    section xaxis();
    #pragma omp
    section yaxis();
    #pragma omp
    section zaxis();
}
```

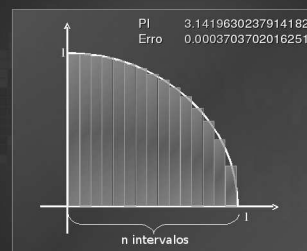
8

Exemplo: Calculando Pi



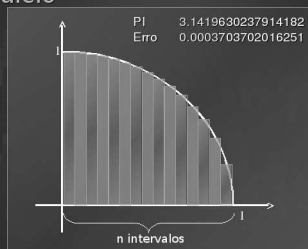
9

Exemplo: Calculando Pi



10

Exemplo: Calculando Pi em Paralelo



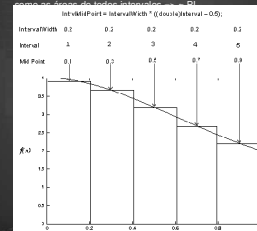
11

Exemplo: Calculando Pi em Paralelo

divida o domínio [0,1] em intervalos
compute a integral no ponto intermediário de cada intervalo:

$$\int \frac{1}{1+x^2} dx = \pi$$

multiplique estes valores pela largura dos intervalos



Message Passing Interface

Na Plataforma Microsoft...

13

Plano

- Introdução a Message Passing Interface
 - O modelo de programação paralela suportado pelo MPI.
 - “MPI for Dummies”: as 6 instruções básicas.
 - MPI avançado:
 - Comunicações não-bloqueantes
 - Comunicações coletivas
 - Tipos de dados definidos pelo usuário
 - MPI-2
 - MPI-IO
 - Criação dinâmica de processos.

Algumas referências sobre MPI

- <http://www.mpi-forum.org>, para a norma MPI.
- Conferência internacional: EuroPVM-MPI (LNCS, Springer)
- Gropp, William *et al.*, **Using MPI**, MIT Press.
- Gropp, William *et al.*, **Using MPI-2**, MIT Press.
- Spir, M. *et al.*, Dongarra, J., **MPI: The**

O que É um programa MPI?

- Um programa MPI = um conjunto de p processos “pesados” que se comunicam através de mensagens.
 - Processo pesado = memória local e privada,
 - Mensagens = rede (Bom... depende...),
 - Tipicamente, os p processos são disparados no início.
- MPI É uma biblioteca, não uma L.P.
- SPMD - Single Program, Multiple Data approach
 - Programa uma vez, mas vários processos executam!
 - Cada processo É identificado por um rank.
- Em tese: escreve-se um programa C, fork-se p processos e os clones usam sockets.

O caso de sucesso do MPI

- Antes do MPI, tinha o PVM (Parallel Virtual Machine)
 - Era na Época das primeiras NoWs.
 - /til, mas faltava padronização.
- 1995 — MPI 1.1
 - 1997 — MPI 1.2
 - 1998 — MPI-2
 - 2000+ — MPI-2 fornecido nas principais distribuições.
- 2 principais distribuições Open Source:
 - OpenMPI (www.open-mpi.org),
 - MPI-CH (www-unix.mcs.anl.gov/mpi/mpi-ch/).
- Muitas distribuições comerciais.
- ... fato que a ampla maioria das aplicações executando em Clusters são baseadas no MPI.
- “MPI is as simple as using 6 functions and as complicated as a user wishes to make it.” [P]

MPI em 6 palavras

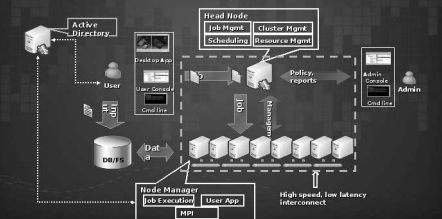
- 1) `MPI_Init( &argc, &argv) // No comment.`
- 2) `MPI_Comm_rank(&r, communicator)`
// retorna o rank in na variável int r
- 3) `MPI_Comm_size(&p, communicator)`
// retorna o número de processos na variável int p
- 4) `MPI_Send( /* muitos args */ )`
- 5) `MPI_Recv( /* quase tantos args */ )`
- 6) `MPI_Finalize() // No comment`

O communicator básico é a constante `MPI_COMM_WORLD`

Executando um programa MPI

`mpirun -np 10 /path/to/mipi-program`

- Um arquivo textual descreve os endereços IP das CPUs que vão executar os processos.
 - Mapeamento round-robin entre as CPUs



Como É uma mensagem MPI?

- Perfil:


```
int MPI_Send(void*, int, MPI_Datatype, int, MPI_Comm).
```
- Chamada típica:


```
MPI_Send(&work, 1, MPI_INT, dest, WORKTAG, MPI_COMM_WORLD);
```
- Manda o conteúdo de um buffer do processo atual para um destinatário.
- O buffer É definido pelos 3 primeiros argumentos:
 - Work (void*): ponteiro para a área onde se encontram os dados.
 - 1, MPI_INT: número e tipo básicos dos elementos no buffer (quase sizeof!).
- A mensagem É identificada por um tag (ver WORKTAG).
 - Deve fechar o tag entre o Send e o Recv.
- O processo destinatário É identificado por seu rank, dentro de um comunicador.

Do lado do destinatário....

- Perfil:


```
int MPI_Recv(void*, int, MPI_Datatype, int, MPI_Comm, MPI_Status*).
```
- Chamada típica:


```
MPI_Recv(&result, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD, &status);
```
- Recebe o conteúdo de uma mensagem e o armazena no buffer 'result'.
- O buffer de recepção É especificado assim como no Send (3 primeiros args).
- Tem um rank da fonte, e um tag.
 - Pode-se usar corrinhas: MPI_ANY_SOURCE, MPI_ANY_TAG.
- O argumento 'status' (o último) provê informação a respeito da mensagem que foi recebida.

Exemplo básico e completo.

Tipos abstratos do MPI

```
void main() {
    int p, r, tag = 103;
    MPI_Status stat;
    double val;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &r);
    if (r == 0) {
        printf("Processor 0 sends a message to 1\n");
        val = 3.14;
        MPI_Send(&val, 1, MPI_DOUBLE, 1, tag, MPI_COMM_WORLD);
    }
    else {
        printf("Processor 1 receives a message from 0\n");
        MPI_Recv(&val, 1, MPI_DOUBLE, 0, tag, MPI_COMM_WORLD, &stat);
        printf("I received the value: %.2f\n", val);
    }
}
```

Algumas observações

- MPI não se limita a programas Mestre/Escravo
 - Padrão frequente: processos de rank par fazem alguma coisa, os ímpares fazem outra coisa.
- Os tags devem ser predefinidos pelo programador.
 - O conjunto (source, tag, dest) identifica a mensagem, então cuidado com possíveis confusões (casts).
- O buffer É uma área de memória contígua
 - Deve-se serializar dados não-contíguos para comunicá-los.
- A mensagem tem tamanho fixo, e o recipiente deve conhecê-lo para efetuar o Recv.
 - Possivelmente, se deve mandar primeiro o tamanho, e depois os dados.

Comm. Bloqueantes vs. Não-bloqueantes

- MPI_Recv(&x,...) É bloqueante.**
 - Quando se executa a próxima instrução, x já tem o valor correto.
- MPI_Send(&x,...) É não-bloqueante**
 - Porém, x É bufferizado, e por isso a próxima instrução pode alterar x sem problema.
 - Send É não-bloqueante... Até estourar o buffer interno!
- Variações não-bloqueantes: MPI_Irecv() e MPI_Isend()**
 - Mesmos argumentos como Recv/Send, com um extra de tipo MPI_Request.
 - O MPI_Request possibilita testar a compleção de comunicações não-bloqueantes.
- Existe também versões explicitamente bufferizadas do Send/Recv: **MPI_Bsend()**.

Testar e esperar comm. não bloqueantes

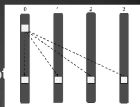
- `MPI_Test(MPI_Request* req, int* flag, MPI_Status* stat)`
 - Seta 'flag' a 0 ou 1, em função do termino da com não-bloqueante descrita por 'req'.
 - Deve-se testar 'flag' depois...
- `MPI_Wait(MPI_Request* req, MPI_Status* stat)`
 - Bloqueia até a comunicação não-bloqueante tenha terminado.
- Prática (muito) comum (sobreposição computação/comunicação) :
 - Dispara uma comm não-bloqueante (e.g recv),
 - (em um laço) executa todo o que se pode fazer sem ter recebido os dados, testando a recepção.
 - Se o laço acabar, bloqueia com Wait.

Comunicações coletivas

- Um dos pontos fortes do MPI: todas as comunicações coletivas são padronizadas.
 - Cabe à implementação da biblioteca otimizar-las com árvores de broadcast, roteamento em hyper-cube, etc...
 - Tão logo se usa mais de 2 processos numa comunicação, nunca use Send/recv!
 - Não reinvente a roda...
- Por definição, todos os processos se sincronizam quando efetuam uma com. coletiva.
 - O MPI não prevê versões não-bloqueantes.
- Pode-se particionar MPI_COMM_WORLD em sub-communicators, para possibilitar comunicações dentro de um sub-grupo de processos.

Algumas comm. coletivas: Bcast

- `int root = 0;`
`double dados[ n ]`
`MPI_Bcast(dados, n, MPI_DOUBLE, root, MPI_COMM_WORLD);`
- O processo de rank 'root' manda n doubles (em dados) para todos os demais.
- TODOS os processos chamam MPI_Bcast
 - Há sincronização global,
 - Internamente, 'root' vai mandar mensagens, os p-1 outros processos vão receber.
 - O buffer ('dados') tem uma semântica diferenciada em função do rank.



MPI_Reduce

- Perfil:
`MPI_Reduce(void*, void*, int, MPI_Datatype, MPI_Op, int, MPI_Comm)`
- Chamada típica:
`double* sum_buf, res;`
`sum_buf = malloc(10*sizeof(double));`
`MPI_Reduce(sum_buf, res, 10, MPI_DOUBLE, MPI_SUM, root, MPI_COMM_WORLD);`
- Aplica um operador de redução no buffer, manda os resultados parciais para 'root', que recupera o resultado global em 'res'.
- O operador pode ser MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_LOR, etc. ...

Calculando Pi (1)

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
    int done = 0, n, myid, numprocs, i, rc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    while (!done) {
        if (myid == 0) {
            printf("Enter the number of intervals:");
            printf("\n(0 quits) ");
            scanf("%d", &n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0) break;
    }

    // Computação
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }

    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myid == 0) {
        printf("pi is approximately %.16f,\n", pi);
        printf("Error is %.16f\n", fabs(pi - PI25DT));
    }
    MPI_Finalize();
    return 0;
}
```

Inicialização

PI "correto"

Inicia MPI

Obtém #processos para esse job

Obtém identificador desse processo

No processo 0, pede ao usuário o número de intervalos

Computação

Envia o #intervalos a todos os

Calculando Pi (2)

```
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
    x = h * ((double)i - 0.5);
    sum += 4.0 / (1.0 + x*x);
}

mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

if (myid == 0) {
    printf("pi is approximately %.16f,\n", pi);
    printf("Error is %.16f\n", fabs(pi - PI25DT));
}
MPI_Finalize();
return 0;
}
```

Calcula as áreas para os intervalos desse processo

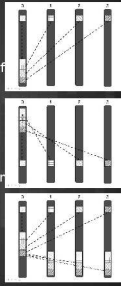
Soma os resultados

Finalização

No processo 0, imprime o valor estimado para pi e o desvio do valor "correto"

Outras comunicações coletivas

- **MPI_Gather,**
 - O processo $\sqrt{root}$ agrega buffers de todos os demais processos (n elementos), em um buffer de $p \times n$ elementos.
- **MPI_Scatter,**
 - Um $p \times n$ buffer está "explodido" entre p processos.
- **MPI_Gatherv, MPI_Scatterv**
 - Versões generalizadas, com buffers de tamanhos diferentes em cada processo.
- **MPI_Allgather**
 - = Gather + Bcast.
- **MPI_Alltoall**
 - O nome diz tudo...



Datatypes de usuário

- Os MPI Datatypes básicos para as comunicações são MPI_INT, MPI_FLOAT, MPI_CHAR, MPI_DOUBLE...
- Como comunicar tipos de dados abstratos?
 - Pode-se usar o "método naive"
 - Ou:
 - Descreve ao MPI sua estrutura de dados,
 - Efetua um "commit" para criar um novo MPI Data_type personalizado,
 - Usa o Data_type como se fosse um Datatype

O método naive (e menos naive)

- Método naive:
 - memcpy() toda a estrutura de dados num buffer de bytes, recupera seu tamanho e manda-o como MPI_CHAR.
 - Pouco elegante, e nem um pouco portátil.
- Solução melhor: usa o MPI para empacotar os dados:
 - MPI_Pack(void* inbuf, int count, MPI_Datatype dtype, void* outbuf, int outcount, int* position, MPI_Comm comm)
 - Empacota os dados em 'inbuf' (= 'count' itens de tipo básico 'dtype'), retorna 'outbuf' e atualiza 'position' para outras chamadas a MPI_Pack().
- Manda o buffer com tipo básico MPI_PACK:
 - MPI_Send(outbuf, MPI_Pack_size(outbuf), MPI_PACK, ...)
- Usa MPI_Unpack(...) depois do MPI_Recv para

MPI-2

- "Nova" norma MPI, que provê novas funcionalidades.
 - Definida em 1998!
 - Apesar de LAM-MPI ter fornecido uma implementação de MPI-2 já em 2000, foi apenas a partir de 2005 que todas as distribuições passaram a implementar MPI-2.
- MPI-2 acrescenta MPI-1.2 com:
 - Criação dinâmica de processos,
 - Facilidades para comunicação entre esses processos;
 - I/O Paralelas;

Criação dinâmica de Processos

- MPI-2 possibilita a criação de processos (MPI) processes durante a execução de um programa MPI.
 - Pode-se "spawnar" apenas programas MPI válidos.
 - Pode-se depois mandar esses processos se comunicar.
- MPI_Comm_spawn(char* command, char* args[], int maxprocs, MPI_Info info, int root, MPI_Comm comm, MPI_Comm* inter_comm, int errors[]);
 - dispara 'maxprocs' novos processos para executar 'command' (= um outro programa MPI).
 - Os 'args' são repassados ao novo programa MPI, como se ele fosse "multiplexado".
 - Em saída, Spawn retorna um 'inter_comm': um novo communicator para interconectar os dois conjuntos de processos.
- O tipo abstrato MPI_Comm foi refinado, em MPI-2, para distinguir intra- vs. Inter-communicators.

Comunicações entre os processos criados

- O mecanismo básico continua sendo a troca de mensagens.
 - O processo pai pode usar o inter_comm retornado por Spawn para "conversar" com os processos filhos.
 - Os processos filhos podem usar MPI_Get_comm para determinar o communicator de seus pais (se têm pais).
- Outra opção É usar MPI_Intercomm_merge para fusionar o inter_comm com o MPI_COMM_WORLD pre-existente (do lado do pai) e o MPI_COMM_WORLD dos filhos recém-criados.

MPI-I/O

- Problema clássico em processamento paralelo: como garantir, no mesmo momento::
 - Acessos paralelos ao sistema de arquivos,
 - Garantir a exclusão mútua necessária?
- Soluções clássicas em MPI:
 - Um processo YrootV centraliza os I/O.
 - Ou cada processo escreve em um arquivo diferente (/tmp).
- Soluções em MPI-2: prover uma nova abstração de arquivos: MPI_File.
 - MPI_File_open — MPI_File_write MPI_File_read — MPI_File_close
- As operações num MPI_File são coletivas (conforme definido pelo communicator).
- Usa MPI_File_view para especificar quais partes de um MPI_File cada processo irá acessar:

Além de MPI...

- MPI É o padrão de fato.
- MPI++ - melhorias dentro do MPI (pesquisa atual)
 - Tolerância a falhas
 - Checkpoint/restarts,
 - ... possível no nível do MPI?
 - ... eficiente no nível do MPI?
 - Migração de processos.
 - Suporte a arquiteturas heterogêneas
 - MPI-G2
- Entretanto, as limitações do MPI são, na verdade, devidas ao seu modelo de programação.
 - "MPI is the assembly language of parallel programming" (B. Grop)
 - Talvez não seja Assembly, mas É no máximo C/Fortran.
 - É OO, Templates, Componentes, usar melhor o Compilador?

Parallel Computing Platform - PCP

Concurrency Runtime
Parallel Extensions to the .NET Framework

39

Qual É o Problema?

- Programação multi-thread É difícil
 - Exige especialistas
 - Padrões paralelos não são bem conhecidos nem fáceis de se implementar
 - Problemas potenciais
 - Concorrência, deadlocks, livelocks, overhead no cache, perda de notificações de eventos, inversão de prioridades,...
- Ao simplificar...
 - Melhores desenvolvedores podem se concentrar em valor para o problema sendo estudado, ao invés de concorrência
 - Como permitir que todos os desenvolvedores escrevam código concorrente?

Exemplo: "Baby Names"

```
IEnumerable<BabyInfo> babies = ...;
var results = new List<BabyInfo>();
foreach (var baby in babies)
{
    if (baby.Name == queryName &&
        baby.State == queryState &&
        baby.Year >= yearStart &&
        baby.Year <= yearEnd)
    {
        results.Add(baby);
    }
}
results.Sort((b1, b2) =>
    b1.Year.CompareTo(b2.Year));
```

41

Paralelismo Manual



```
IEnumerable<BabyInfo> babies = ...;
var results = new List<BabyInfo>();
int partition = Environment.ProcessorCount;
int newCount = results.Count;
var partitioner = new Partitioner(babies);
var parallelQuery = partitioner.GetPartitions(partition);
foreach (var partition in parallelQuery)
{
    done = new ManualResetEvent(false);
    partition.Count = 1;
    newCount = new List<BabyInfo>();
    {
        if (baby.Name == queryName &&
            baby.State == queryState &&
            baby.Year >= yearStart &&
            baby.Year <= yearEnd)
        {
            newCount.Add(baby);
        }
    }
    done.Set();
}
results.AddRange(newCount);
results.Sort((b1, b2) =>
    b1.Year.CompareTo(b2.Year));
finally {
    partition.Dispose();
}
```

Conhecimento de sinc.

Trabalhos Ineficientes

Perda da simplicidade do foreach

Agregação manual

Tricks

Perda da razão de threads

Paralelização pesada

Ordenação não paralela

42

O que É a Plataforma de Computação Paralela?

- Parte da Iniciativa de Computação Paralela da Microsoft
- Objetivo
 - Levar computação paralela aos desenvolvedores na plataforma Microsoft (C++, .NET)
- Por quê?
 - Atender à realidade de hardware
 - Oferecer experiências ricas

Parallel Computing Initiative



Algumas Formas de Paralelismo

Tarefas
Dados (Declarativo e Imperativo)

45

Paralelismo de Tarefas

- Tanto em .NET quanto em C++
 - Task Parallel Library (.NET)
 - Parallel Pattern Library (C++)
- Principais características
 - Abstrações de tarefas OO
 - Ciclo de vida lógico e transições de estado bem definidas
 - Concorrência estruturada (relações parent/child)
 - Cancelamento
 - Tratamento comum de exceções
 - Fluxo de dados

Paralelismo de Dados Imperativo

- TPL e PPL
 - Laços for paralelos (intervalos de inteiros e iteradores)
 - Recursão (limites dinâmicos)
 - Reduções
 - Ordenações In-place
- Ex., laço "for" simples

```
Parallel.For(0, N, i => { ... });
parallel_for(0, N, [&](int i) { ... });
```
- Ex., também sobre enumerações .NET e iteradores C++

```
Parallel.ForEach(e, t => { ... });
parallel_for_each(e->begin(), e->end(), [&](T * t) { ... });
```

Paralelismo de Dados Declarativo

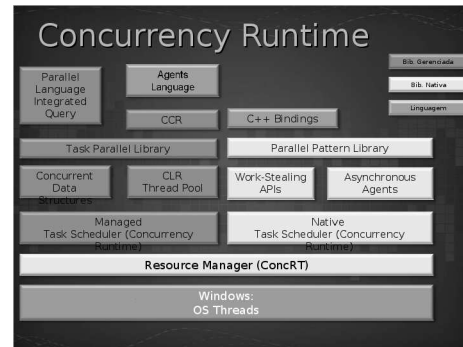
- PLINQ == Parallel (P) LINQ-to-Objects (LINQ)
 - LINQ veio com .NET Framework 3.5
 - Operadores SQL-like para unificar acesso a dados (BD, in-memory e XML)
 - Pode realizar consultas heterogêneas (ex., join entre XMS e in-memory)
 - C# e VB: extensões de sintaxe
 - PLINQ auto-paraleliza in-memory e XML
- Ex., filter, project, sort

```
int[] arr = ... some big array ...;
var q = from x in arr
        where p(x) // p: predicate/filter
        orderby k(x) // k: key selection
        routine
        select f(x); // f: data transformation
float[] out = q.ToArray(); // Execute!
```

Concurrency Runtime

Parallel Pattern Library (PPL)

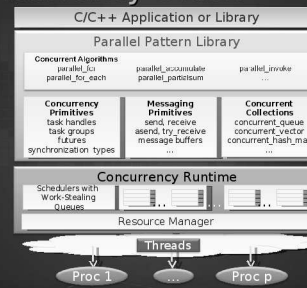
49



Concurrency Runtime

- Concurrency Runtime
 - Mapeia tarefas (lógico) para recursos físicos (threads e processadores)
 - Data structures mirror hardware topology: packages, nodes, and caches --- to optimize for locality
- C++ runtime exposto publicamente
 - OpenMP, ...
- .NET runtime usado por abstrações

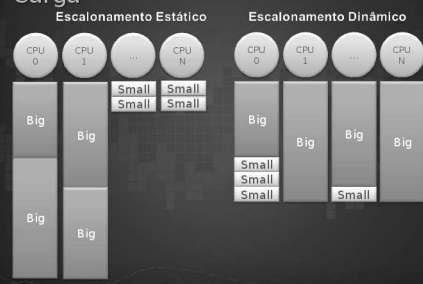
Concurrency Stack



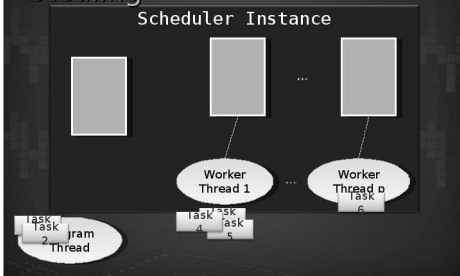
Concurrency Runtime

- Escalonador com baixo overhead
 - Algoritmos eficientes para execução de tarefas
 - Bloqueios cooperativos
 - Múltiplas instâncias do escalonador
 - Escalonamento baseado em políticas
- Gerenciamento de Recursos
 - Comum
 - Balanceamento de carga dinâmico

Importância de Balanceamento de Carga



Como funciona o Work Stealing



Parallel Pattern Library

- Facilita expressar concorrência em aplicações C++
 - Concurrency Primitives
 - task_handle
 - task_group
 - future <T>
 - Synchronization primitives
 - Messaging Primitives
 - AgentTask
 - send, receive
 - asend, try_receive
 - ISource, ITarget
 - unbounded_buffer
 - overwrite_buffer
 - single_assignment
 - call, transform, timer
 - Algorithms
 - parallel_for
 - parallel_for_each
 - parallel_invoke
 - parallel_accumulate
 - parallel_partial_sum
 - Concurrent Collections
 - concurrent_queue
 - concurrent_vector
 - concurrent_hash_map

Exemplo: Multiplicação de Matrizes

```
void MatrixMult(
    int size, double** m1, double** m2, double** result)
{
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            result[i][j] = 0;
            for (int k = 0; k < size; k++) {
                result[i][j] += m1[i][k] * m2[k][j];
            }
        }
    }
}
```

57

Parallel Pattern Library

```
void MatrixMult(
    int size, double** m1, double** m2, double** result)
{
    parallel_for(0, size, 1, [&](int i) {
        for (int j = 0; j < size; j++) {
            result[i][j] = 0;
            for (int k = 0; k < size; k++) {
                result[i][j] += m1[i][k] * m2[k][j];
            }
        }
    });
}
```

58

Concurrency Primitives

```
class task_handle {
public:
    template <class Func>
    task_handle(Func & f);
};

class task_group {
public:
    task_group();
    template <class ExceptionFunc>
    task_group(ExceptionFunc Fn);
    template <class Func>
    void run(Func Fn);
    void run(task_handle * t);
    void interrupt();
    task_status wait();
};
```

59

Exemplo

```
void quicksort(int * a, int n) {
    if (n <= 1) return;
    int s = partition(a,n);
    quicksort(a,s);
    quicksort(a+s,n-s);
}

void quicksort(int * a, int n) {
    if (n <= 1) return;
    int s = partition(a,n);
    task_group g;
    g.run([&]{quicksort(a,s);});
    g.run([&]{quicksort(a+s,n-s);});
    g.wait();
}
```

60

Outro Exemplo

```
solution computeResult() {
    task_group tg;

    future<solution> solution1 =
        ( &tg, []{ return heuristic1(); } );
    future<solution> solution2 =
        ( &tg, []{ return heuristic2(); } );

    solution result =
        (solution1 || solution2).value();

    tg.interrupt();
    return result;
}
```

61

Concurrent Algorithms

```
void quicksort(int * a, int n) {
    if (n <= 1) return;
    int s = partition(a,n);
    quicksort(a,s);
    quicksort(a+s,n-s);
}

void quicksort(int * a, int n) {
    if (n <= 1) return;
    int s = partition(a,n);
    parallel_invoke(
        [&]{quicksort(a,s);},
        [&]{quicksort(a+s,n-s);});
}
```

62

Exemplo

```
V momentum(V& m, const Planet& p) {
    return m + p.mass * p.vel;
}

void offset_momentum(vector<Planet& b) {
    V z(0.0, DIM);
    b[0].vel = (-1/solar_mass) * accumulate(
        b.begin(), b.end(), z, momentum);
}

void offset_momentum(vector<Planet& b) {
    V z(0.0, DIM);
    b[0].vel = (-1/solar_mass) * parallel_accumulate(
        b.begin(), b.end(), z, momentum);
}
```

63

Parallel Extensions to the .NET Framework

PLINQ
Task Parallel Library
Coordination Data Structures

64

Parallel Extensions to the .NET Framework

- O que É?
 - .NET library
 - Sem mudanças no compilador
 - Leve, execução em modo usuário
 - Paralelismo imperativo e declarativo, de dados e tarefas
 - Paralelismo de dados declarativo (PLINQ)
 - Paralelismo imperativo de dados e tarefas (Task Parallel Library)
 - Construção para coordenação/sincronização (Coordination Data Structures)
 - Modelo comum para tratamento de exceções
- Qual o objetivo?
 - Fornecer recursos de paralelismo em qualquer linguagem .NET
 - Menor complexidade, melhor produtividade
 - Paralelização indo de especialistas em concorrência para especialistas em domínio
- Quando?
 - **June 2008 CTP disponível!** (<http://msdn.microsoft.com/concurrency>)

Exemplo: “Baby Names”

```
IEnumerable<BabyInfo> babies = ...;
var results = new List<BabyInfo>();
foreach (var baby in babies)
{
    if (baby.Name == queryName &&
        baby.State == queryState &&
        baby.Year >= yearStart &&
        baby.Year <= yearEnd)
    {
        results.Add(baby);
    }
}
results.Sort((b1, b2) =>
    b1.Year.CompareTo(b2.Year));
```

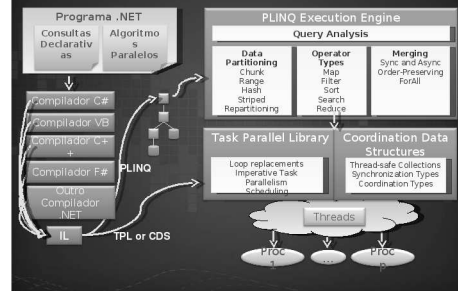
66

Solução com LINQ

```
var results = from baby in babies.AsParallel()
              where baby.Name == queryName &&
                    baby.State == queryState &&
                    baby.Year >= yearStart &&
                    baby.Year <= yearEnd
              orderby baby.Year ascending
              select baby;
```

67

Arquitetura



Parallel LINQ (PLINQ)

- Permite que desenvolvedores LINQ (Language Integrated Query) utilizem hardware paralelo
 - Suporte a todos os operadores de queries do .NET
 - Mais alguns métodos específicos do PLINQ
 - Abstrai detalhes de paralelização
 - Funciona para IEnumerable<T>

Escrevendo uma Query PLINQ

- Dois modos
 - Extensões de sintaxe do C# e Visual Basic


```
var q = from x in Y.AsParallel() where p(x) orderby x.f1 select x.f2;
```
 - APIs
 - Como extensões em IEnumerable<T>


```
var q = Y.AsParallel().Where(x => p(x)).OrderBy(x => x.f1).Select(x => x.f2);
```
 - Classe System.Linq.ParallelEnumerable


```
var q = ParallelEnumerable.Select(
    ParallelEnumerable.OrderBy(
        ParallelEnumerable.Where(Y.AsParallel(), x => p(x)),
        x => x.f1),
        x => x.f2);
```
- A implementação da API faz o trabalho real
 - O compilador converte o primeiro para o segundo

Particionamento

- A entrada É particionada em p subconjuntos disjuntos
- Os operadores são replicados nas partições
- Exemplo


```
(from D in Database
 where p(x)
 select x).Sum();
```
- Cada partição É executada em completo (quase) isolamento



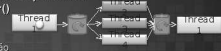
Particionamento, cont.

- Tipos de particionamento
 - Chunk
 - Funciona com qualquer IEnumerable<T>
 - Compartilha um enumerador: chunks tratados sob demanda
 - Range
 - Funciona apenas com IList<T>
 - Entrada dividida em ranges contíguos, uma por partição
 - Stride
 - Funciona apenas com IList<T>
 - Os elementos são enviados às partições no modelo round-robin
 - Hash
 - Funciona com qualquer IEnumerable<T>
 - Os elementos são enviados às partições baseado em código hash
- Reparticionamento às vezes É necessário

Merging

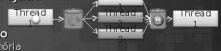
- Pipelined: *threads de consumo separadas*

- Default para GetEnumerator()
- É portanto para foreach
- Acesso a dados disponível
- Mais overhead de sincronização



- Stop-and-go: *quem consome ajuda*

- Sorts, ToArray, ToList, GetEnumerator(false), etc.
- Reduz mudanças de contexto
- Mais alta latência e mais memória



- Inverted: *sem merge*

- Método ForAll
- Mais eficiente
- Mas nem sempre aplicável



Task Parallel Library Laços

- Laços são consumidores de recursos

```
for (int i = 0; i < n; i++) work(i);
...
foreach (T e in data) work(e);
```

- Podem ser paralelizados quando iterações forem independentes

- Corpo não depende de estado mutável
- Ex. Variáveis estáticas sendo escritas para variáveis locais para

```
Parallel.For(0, n, i => work(i));
...
Parallel.ForEach(data, e => work(e));
```

- Síncronos

- Todas as iterações terminam

Task Parallel Library

- Sequência de comandos

```
StatementA();
StatementB;
StatementC();
```

- Quando independentes, podem ser paralelizados

```
Parallel.Invoke(
    () => StatementA(),
    () => StatementB,
    () => StatementC() );
```

- Síncrono (assim como nos laços)

Coordination Data Structures

- Usadas pelo PLINQ e TPL

- Resolve alguns problemas de concorrência

- Thread-safe collections
 - ConcurrentStack<T>
 - ConcurrentQueue<T>
 - ...

- Locks

- ManualResetEventSlim
- SemaphoreSlim
- SpinLock
- SpinWait
- ...

- Work exchange

- BlockingCollection<T>
- ...

- Initialization

- LazyInit<T>
- WriteOnce<T>
- ...

- Phased Operation

- CountdownEvent
- ...

Sugestões de Desempenho

- Grandes conjuntos de dados e/ou Compute intensive
 - Trabalho deveria ser de ao menos milhares de ciclos
 - Meça e otimize como necessário

- Não paralelize à toa

- ... leve, mas ainda existe alocação de objetos,...

- Paralelize os laços externos apenas

- A menos que o número de iterações não justifique
- Nesse caso, considere paralelizar os laços internos ou ambos

- Prefira isolamento ao invés de sincronização

- Sincronização == Escalável
- Tente evitar dados compartilhados

- Tenha expectativas realistas

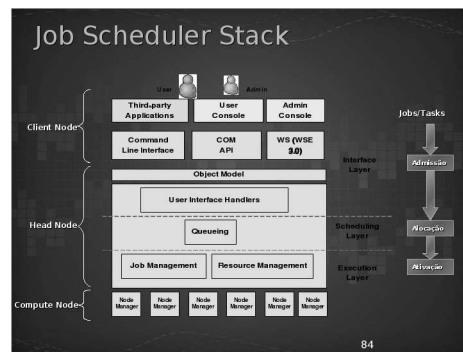
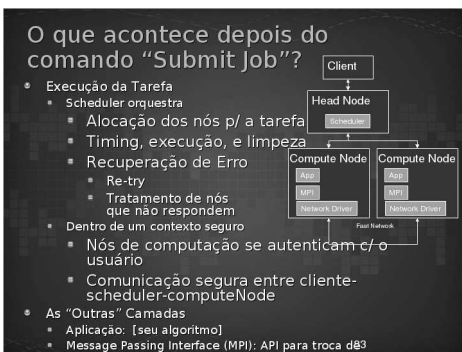
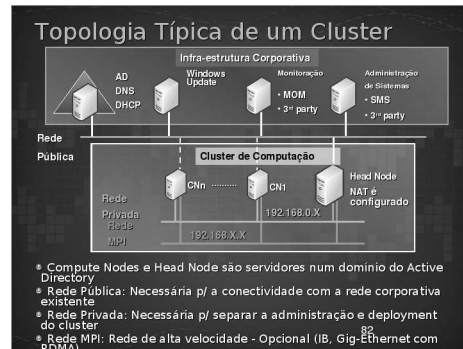
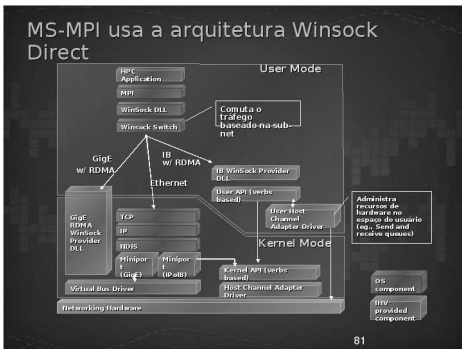
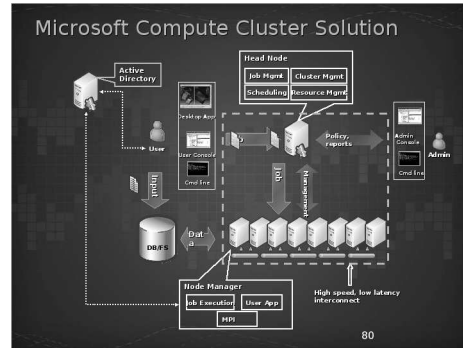
- Lei de Amdahl
- Velocidade É limitada pela quantidade de computação sequencial

Conclusão

- Programação Multicore já está presente

- Alternativas:

- Em C/C++: OpenMP e PPL
- Em .NET: Parallel Extensions (PLINQ e TPL)
- Clusters e/ou Multicore: MPI



Segurança Fim a Fim

