O padrão OpenMP para computação paralela

Danilo Ehrhardt Ferreira Bento
<danilo@ime.usp.br>

Orientador: Marco Dimas Gubitoso
<gubi@ime.usp.br>

Seminários de Sistemas de Software

Instituto de Matemática e Estatística
Universidade de São Paulo

13 de novembro de 2007

Introdução

- Artigo ”Attack of the Killer Micros” – Wall Street Journal, 1998 – dizia que sistemas computacionais feitos de diversos processadores de baixo custo poderiam tornar os supercomputadores obsoletos.
- 400 computadores pessoais poderiam equiparar-se a um supercomputador ?

Introdução

- Alguns problemas:
  - troca de informações:
    - velocidade;
    - latência.
  - mecanismos de controle:
    - concorrência;
    - coerência.
  - programas:
    - reengenharia;
    - novos algoritmos.

Introdução

- Os microprocessadores ganharam grande desempenho aproveitando-se da evolução dos supercomputadores, seus estudos e a quebra de barreiras técnicas que conseguiram.
- Implementação destas melhorias em um curto espaço de tempo nos microprocessadores.

Introdução

- A evolução dos microprocessadores está também associada à demanda por mais desempenho, impulsionada por:
  - Gráficos em 3D;
  - Interfaces gráficas com o usuário;
  - Multimídia;
  - Jogos.

Introdução

- Com a evolução dos microprocessadores, um microprocessador de hoje pode ser comparado a centenas de microprocessadores do passado, atingindo mais desempenho do que era necessário para a substituição de alguns modelos de supercomputadores por um computador pessoal.

Introdução

- Porque não agrupar diversos microprocessadores modernos em um sistema?
  - SMP, ”Multithread”, ”Multicore”:
    - troca de informações através de memória compartilhada;
    - mecanismos de controle de coerência.
  - programas:
    - mecanismos de controle de concorrência;
    - reengenharia;
    - novos algoritmos.

Padrão

- A API OpenMP especifica:
  - uma coleção de diretivas de compilador;
  - uma biblioteca de rotinas;
  - variáveis de ambiente.
- Um modelo portável de programação paralela para arquiteturas de memória compartilhada.

Padrão

- Usado para especificar paralelismo em programas escritos nas linguagens C, C++ e Fortran:
  - ISO/IEC 9899:1990 (C90);
  - ISO/IEC 9899:1999 (C99);
  - ISO/IEC 14882:1998 (C++);
  - ISO/IEC 1539:1980 (Fortran 77);
  - ISO/IEC 1539:1991 (Fortran 90);
  - ISO/IEC 1539-1:1997 (Fortran 95).
- Compiladores de diversos fabricantes dão suporte a API OpenMP.

Padrão

- Modelo de execução paralela “fork-join”.
- Projetado para programas que executam corretamente de forma seqüencial ou paralela.
  - facilita a reengenharia de programas.
- Pode ser usado em programas que sejam estritamente paralelos em sua concepção.

Padrão

- Programas iniciam com apenas uma linha de execução (“thread”), chamada “thread” inicial;
- Programa segue a execução da “thread” inicial, como uma execução seqüencial, até encontrar uma região de execução paralela;
- Ao término da região paralela, somente a “thread” principal segue a execução das instruções seguintes.

Padrão

- OpenMP aproveita código existente, paralelização pode ser implementada de forma incremental;
- MPI requer reengenharia completa dos programas, pode ser usado em sistemas sem memória compartilhada;
- Pthread requer certa reengenharia, API requer uma certa iteração em mais baixo-nível.

API

- Diretivas de compilador:
  - C/C++:

    #pragma omp
  - Fortran:

    !$omp

API

- A construção **parallel** é a construção fundamental que inicia uma execução paralela.
  - C/C++:

    #pragma omp parallel [cláusula[[,]cláusula] ...] nova-linha
    bloco-estruturado
  - Fortran:

    !$omp parallel [cláusula[[,] cláusula]...]
    bloco-estruturado
    !$omp end parallel

API

- A construção **parallel** possui as seguintes cláusulas:
  - C/C++:
    - **if(**expressão-condicional**)**
    - **private(**lista**)**
    - **firstprivate(**lista**)**
    - **default(shared | none)**
    - **shared(**lista**)**
    - **copyin(**lista**)**
    - **reduction(**operador: lista**)**
    - **num_threads(**inteiro**)**

API

- Fortran:
  - **if(**expressão-condicional**)**
  - **private(**lista**)**
  - **firstprivate(**lista**)**
  - **default(private | shared | none)**
  - **shared(**lista**)**
  - **copyin(**lista**)**
  - **reduction(**operador: lista**)**
  - **num_threads(**inteiro**)**

API

- Exemplo em C:

```
#include <stdio.h>

int main() {
#pragma omp parallel
    printf(”Hello world!\n”);

    return 0;
}
```

```
$ gcc-4.2 -fopenmp \
> -o hello hello.c
$ ./hello
Hello world!
Hello world!
$
```

API

- Algumas subrotinas da bilbioteca:
  - omp_set_num_threads
  - omp_get_num_threads
  - omp_get_max_threads
  - omp_get_thread_num
  - omp_get_num_procs
  - omp_in_parallel

API

- Exemplo em C:

```
#include <stdio.h>
#include <omp.h>

int main() {
        int tid;
        if(omp_in_parallel()) printf("oops... parallel\n");
#pragma omp parallel private(tid)
        {
                tid = omp_get_thread_num();
                if(tid == 0) {
                        int tnum = omp_get_num_threads();
                        printf("%d threads.\n", tnum);
                }
```

API

- Exemplo em C (continuação):

```
            printf("Hello world! Thread %d.\n", tid);
            if(omp_in_parallel()) printf("parallel\n");
        }
        if(omp_in_parallel()) printf("oops again... parallel\n");

        return 0;
    }
```

API

- Exemplo em C (saída em tela):

```
$ gcc-4.2 -fopenmp -lgomp hello2.c -o hello2
$ ./hello2
2 threads.
Hello world! Thread 0.
parallel
Hello world! Thread 1.
parallel
$
```

API

- A construção ”loop” é uma construção para divisão de trabalho.
  - C/C++:

    #pragma omp for [cláusula[[,]cláusula] ...] nova-linha

    bloco-estruturado
  - Fortran:

    !$omp do [cláusula[[,] cláusula]...]

    bloco-estruturado

    !$omp end do [nowait]

API

- Uma cláusula importante na construções de ”loop” é a cláusula **schedule**:

  schedule(tipo[, tamanho])

  - O tipo define como o trabalho é dividido e pode ser:
    - static – uniformemente entre as ”threads”;
    - dinamic – dinâmicamente entre as ”threads”;
    - guided – dinâmicamente entre as ”threads”, mas com tamanhos cada vez menores;
    - runtime – decidido em tempo de execução.

API

- Exemplo em C:

```
#include <stdio.h>
#include <omp.h>
int main () {
        int i;
#pragma omp parallel
#pragma omp for
        for (i=1; i<5; i++) /* i is private by default */
                printf("i=%d thread=%d\n", i, omp_get_thread_num());
        return 0;
}
```

API

- Exemplo em C:

```
#include <stdio.h>
#include <omp.h>
int main () {
        int i;
#pragma omp parallel for
        for (i=1; i<5; i++) /* i is private by default */
                printf("i=%d thread=%d\n", i, omp_get_thread_num());
        return 0;
}
```

API

- Exemplo em C:

```
$ gcc-4.2 -fopenmp -lgomp ompfor.c -o ompfor
$ ./ompfor
i=3 thread=1
i=4 thread=1
i=1 thread=0
i=2 thread=0
$
```

API

- Variáveis de ambiente:
  - OMP_SCHEDULE
  - OMP_NUM_THREADS
  - OMP_DYNAMIC
  - OMP_NESTED

API

- Exemplo:

```
$ OMP_NUM_THREADS=4 ./ompfor
i=1 thread=0
i=4 thread=3
i=2 thread=1
i=3 thread=2
$
```

API

- A construção **sections** é uma construção para divisão de trabalho, onde cada **section** é executada em uma ”thread”.
  - C/C++:

```
#pragma omp sections [cláusula[ [, ]cláusula] ...] nova-linha
{
    [#pragma omp section nova-linha]
        bloco-estruturado
    [#pragma omp section nova-linha
        bloco-estruturado]
}
```

API

- Fortran:

!$omp sections [cláusula[[,] cláusula]...]
[!$omp section]
bloco-estruturado
[!$omp section
bloco-estruturado]
!$omp end sections [nowait]

API

- Exemplo em C:

```
#include <stdio.h>
#include <omp.h>
int main () {
      int i;
#pragma omp parallel sections private(i)
      {
#pragma omp section
      for (i=1; i<5; i++) printf("loop=1 i=%d thread=%d\n", i,
         omp_get_thread_num());
#pragma omp section
      for (i=1; i<5; i++) printf("loop=2 i=%d thread=%d\n", i,
         omp_get_thread_num());
      }
      return 0;
}
```

API

- Exemplo (saída):

```
$ OMP_NUM_THREADS=4 ./ompforsections
loop=1 i=1 thread=0
loop=1 i=2 thread=0
loop=1 i=3 thread=0
loop=1 i=4 thread=0
loop=2 i=1 thread=2
loop=2 i=2 thread=2
loop=2 i=3 thread=2
loop=2 i=4 thread=2
$
```

API

- Outras construções:
  - single – executada em uma ”thread”;
  - master – executada na ”thread” principal;
  - critical – uma ”thread” por vez;
  - barrier – ”threads” aguardam outras chegarem a este ponto antes de continuarem a execução;
  - atomic – evita que uma região de memória seja atualizada simultâneamente por mais multiplas ”threads”;
  - flush – força o sincronismo da região de memória da ”thread” com a memória principal.

Implementações

Fabricante	Plataforma
Fujitsu/Lahey	Intel Linux
Fujitsu/Lahey	Fujitsu Solaris
HP	PA-RISC/Itanium HP-UX
HP	Tru64
IBM	PowerPC AIX/Linux
Intel	IA32/Itanium Linux/Windows
Intel	Intel Mac OS X
KAI Software Lab	Intel Linux/Windows
PGI	Intel Linux/Solaris/Windows
SGI	MIPS IRIX
Sun Microsystems	SPARC/x86/x64 Solaris
Crescent Bay Software	IA32/Itanium/x64/PowerPC Linux
Crescent Bay Software	SGI Altix
Crescent Bay Software	Mac OS X
gcc	Diversas

Referências

- DOWD, Kevin e SEVERANCE, Charles R. **High Performance Computing**. 2. ed. O'Reilly, 1998.
- OpenMP Architecture Review Board. **OpenMP Application Program Interface**: Version 2.5. 2005. Disponível em <http://www.openmp.org/drupal/mp-documents/spec25.pdf>. Acesso em 26/09/2007.
- OpenMP Architecture Review Board. **Compilers and Platforms**. 2007. Disponível em <http://www.openmp.org/drupal/node/view/9#Compilers>. Acesso em 26/09/2007.
- GCC team. **GCC 4.2 Release Series Changes, New Features, and Fixes**. 2007. Disponível em <http://gcc.gnu.org/gcc-4.2/changes.html>. Acesso em 29/09/2007.