Software in Threads aufteilen

Einfacher Parallelisieren mit OpenMP

Um die Leistungsfähigkeit von Mehrkernrechnern zu nutzen, ist neben dem Multitasking von nicht parallelisierten Applikationen durch das Betriebssystems zusätzlich auch die individuelle Parallelisierung der zu benutzenden Applikationen notwendig. Somit muss der Softwareentwickler seine existierenden Programme in Threads aufteilen oder von vornherein bei der Neuentwicklung einen parallelen Programmieransatz wählen.

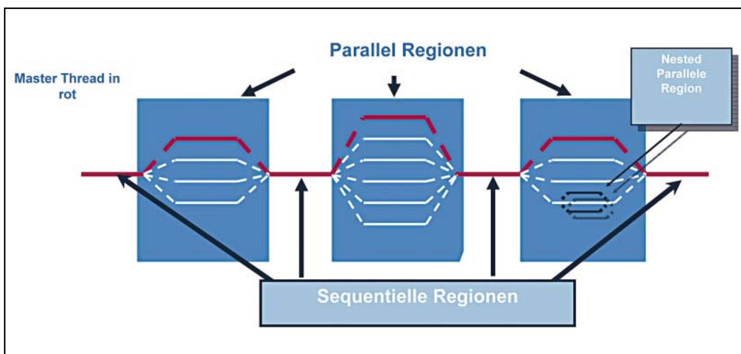


Bild 1: OpenMP Konzept des Join-Fork Modells

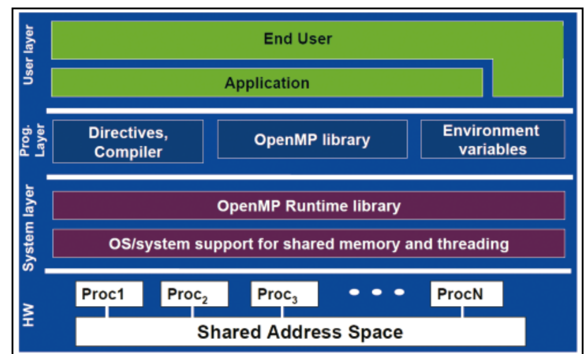


Bild 2: OpenMP innerhalb des Solution Stacks

Es gibt verschiedene Vorgehensweisen, um ein Programm zu parallelisieren, die sich zum Teil erheblich in Schwierigkeitsgrad, Umsetzungsdauer (von seriell auf parallel), Fehleranfälligkeit, Debugging-Aufwand, Lesbarkeit und Leistungsgewinn unterscheiden. Am weitesten ist die Anzahl der Parallelisierungsalternativen beim ‚nativen' Programmieren (im Vergleich zum ‚Managed'-Programmieren) gediehen.

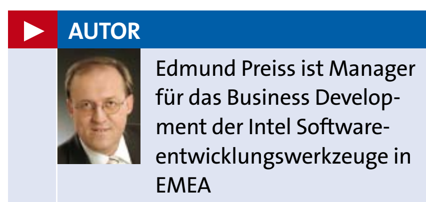
Bei der nativen Programmierung gibt es Konzepte mit relativ niedriger Abstraktionsebene. Dazu gehören die ‚Low-Level'-Threading-APIs (PThreads im Unix-Umfeld und WinThreads im Windows-Umfeld). Diese Parallelisierungs-APIs sind sehr zeitaufwendig zu programmieren und extrem fehleranfällig, was zu hohen Debugging-Zeiten führt. In einer zweiten Gruppe gibt es parallele Programmiermodelle mit einem höheren Abstraktionsgrad. Dazu gehören unter anderem die TBBs (Threading Building Blocks), eine Open-Source-Template-Bibliothek von Intel, die (in Zukunft verfügbare) Parallel-Pattern-Library (PPL) von Microsoft sowie OpenMP, die von diversen Anbietern erhältlich ist.

OpenMP ist ein standardisierter Parallelisierungsansatz aus dem Jahr 1997, der von mehreren Herstellern aktiv unterstützt wird. OpenMP 3.0 ist die neueste Standardisierungvariante, die unter anderem in Intels Fortran- und C++ Compiler integriert ist.

OpenMP eignet sich besonders für einen inkrementellen Parallelisierungsansatz in Shared-Memory-Systemen (SMP). Inkrementell bedeutet hierbei, dass ein existierendes serielles Programm in ein paralleles umgewandelt wird. Der Zeitaufwand, OpenMP zu erlernen und damit inkrementell zu entwickeln, ist relativ gering.

Die Lesbarkeit der parallelisierten Anwendung ist recht hoch, was wichtig ist, wenn mehrere Entwickler an einem Programm während dessen Lebenszyklus' arbeiten. Auch der Debugging-Aufwand und die Fehlerkorrektur sind recht überschaubar, obwohl bei parallelen Programmen nichtdeterministische Fehlerquellen auftauchen (Deadlocks und Dataraces).

AUTOR



Edmund Preiss ist Manager für das Business Development der Intel Softwareentwicklungswerkzeuge in EMEA

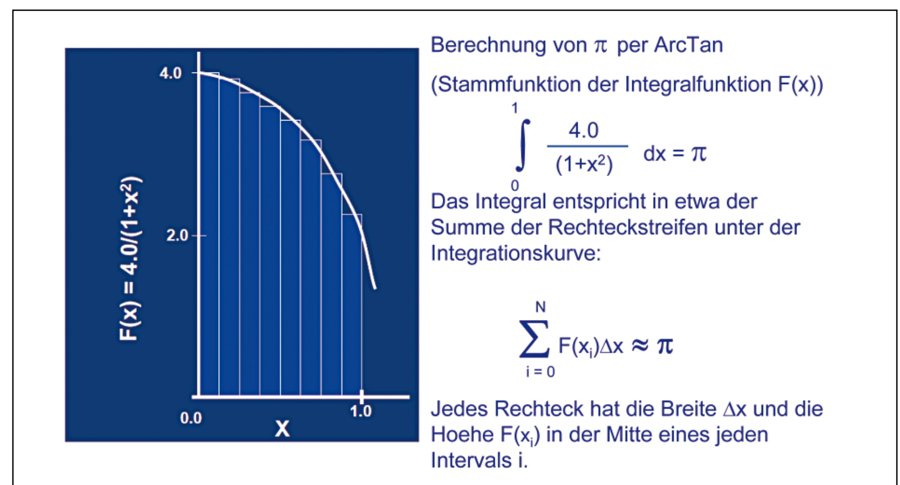


Bild 3 Numerische Integration von π

Ein weiterer Vorteil besteht darin, dass man mit minimalem Aufwand zwischen dem seriellen und dem parallelen Debuggen umschalten kann; OpenMP wird quasi ausgeschaltet. Damit lassen sich die eben genannten nichtdeterministischen Fehlertypen für Debuggingzwecke temporär ausblenden, um so wie bei einem seriellen Programm die Funktionalität in einem deterministischen Umfeld zu testen. Dies ist ein nicht zu unterschätzender Vorteil gegenüber anderen Methoden wie bei der TBB oder bei der PPL. OpenMP wird in der Regel nicht so hochperformanten parallelen Code generieren wie beispielsweise TBB. Dennoch sind die Geschwindigkeitssteigerungen typischerweise imposant und sollten deshalb nicht außer Acht gelassen werden.

OpenMP-Programmierungmodell

Bild 1 stellt das zugrunde liegende Parallelisierungskonzept dar. Gemäß Amdahl's-Law ist der Entwickler gefordert, parallelisierbaren Programmcode zum Beispiel mit einem Analystetool wie etwa Vtune im seriellen Programm zu identifizieren. Diese Bereiche lassen sich dann typischerweise durch OpenMP-Programmdirektiven per Join-Fork-Mechanismus parallel aufspalten bis sie dann zu einem Synchronisierungspunkt wieder in einen Masterthread zusammenlaufen. Zu jedem Zeitpunkt gibt es einen Masterthread. Nur während der parallelen Ausführung wird parallel in mehrere Threads verzweigt. Jeder Thread kann weiterverschachtelt werden (Nested-Modus).

Bild 2 zeigt das OpenMP Programmiermodel in Stack-Ansicht. Die OpenMP-Runtime-Bibibliothek setzt auf das Betriebssystem-API auf. Darüber liegen 3 Bereiche, die der Programmierer in seinem Code verwenden kann. Hierbei handelt es sich um die Direktiven (auch Pragmas genannt), die OpenMP-Bibliothek und die Umgebungsvariablen.

```
static long num_steps = 100000;
double step;

void main ()
{
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;

    for (i=1;i<= num_steps; i++){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
    }

    pi = step * sum;
}
```

Bild 4: Programm 1 – Numerische Kalkulation von π; Serielle Lösung

```
#include <omp.h>
static long num_steps = 100000;
double step;

#define NUM_THREADS 2

void main ()
{
  int i; double x, pi, sum = 0.0;
  step = 1.0/(double) num_steps;

  omp_set_num_threads(NUM_THREADS);

  #pragma omp parallel for reduction(+:sum) private(x)
  for (i=1;i<= num_steps; i++){
            x = (i-0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
  }
  pi = step * sum;
}
```

Bild 5: Programm 2 – Numerische π-Berechnung mit OpenMP; Paralleler Programmieransatz

Die Verwendung dieser 3 Bereiche und einige der Möglichkeiten und Grenzen von OpenMP soll am Beispiel der Kalkulation der Zahl π gezeigt werden, deren Algorithmus in **Bild 3** beschrieben ist. Programmbeispiel 1 zeigt in **Bild 4** die numerische serielle Programmumsetzung der π-Kalkulation.

Eine der möglichen OpenMP-Implementierungen wird in Programmbeispiel 2 (**Bild 5**) dargestellt. Der Code basiert übrigens auf einem C++-Compiler. Die OpenMP-Anweisungen sind in Fortran sehr ähnlich und können aus einer OpenMP-Beschreibung entnommen werden. Die vorgenommenen OpenMP-spezifischen Änderungen des seriellen Programms sind fettgedruckt und in Blau dargestellt.

Die erste Anweisung #include <omp.h> bindet die OpenMP Bibliothek ein.

Die zweite hinzugefügte Anweisung #define NUM_THREADS 2 definiert eine Umgebungsvariable. Mit omp_set_num_threads(NUM_THREADS) wird die Umgebungsvariable im Programmablauf aufgerufen. Mit diesen beiden Anweisungen wird die Anzahl der maximal möglichen Threads auf 2 festgelegt. Selbst auf einem Quadcore-Rechner werden dann nur noch 2 statt idealerweise 4 Threads verwendet, was den Quadcore nur zur Hälfte auslasten würde. Es gibt aber Umstände, bei denen die gewählte Vorgehensweise gewünscht ist.

Kern der OpenMP-Nutzung sind die Compiler-Direktiven, die auch als Pragma-Anweisungen bezeichnet und beim C++ Compiler mit #pragma eingeleitet werden. So etwa erlaubt eine #pragma omp parallel Anweisung, dass die nachfolgende Anweisungen (mit { } eingeklammert) als Threads parallel auf den vorhandenen Rechnerkernen ausgeführt werden.

In dem Beispiel stellt die For-Schleife den Programmkern dar, der gleichzeitig der zeitintensivste Programmteil ist und damit für die Parallelisierung am besten geeignet erscheint.

Neuerungen und Ausblick

OpenMP 3.0 führt gegenüber der bis Ende letzten Jahres gültigen Version 2.5 das Prinzip der Task-Verarbeitung (Task Queing) standardmäßig ein. Dies ist eine wesentliche Neuerung gegenüber der 2.5 Spezifikation. Außerdem ist das Affinity-Feature hinzugekommen. Darunter versteht man, dass man einen Thread einem bestimmten Core zuordnet. (av)