

Herbert Schmid

Siebte Generation

Windows-C++-Compiler von Microsoft und Intel

Die aktuelle siebte Compiler-Generation scheint uneins über die zukünftige Richtung. Microsoft zielt auf die zunehmende Unabhängigkeit vom Prozessor – Intel dagegen will möglichst nah ran, um noch das letzte Quäntchen an Leistung herauszuholen. Dabei soll auch die automatische Parallelisierung mithelfen.

Nicht nur mit seiner .NET-Strategie sucht Microsoft die Unabhängigkeit vom jeweiligen Prozessor. Auch das 64-Bit-Windows wirft seine Schatten in die 32-Bit-Welt, vor allem da sich neben die Fassung für Intels Itanium auch eine für AMDs Hammer gesellen wird. Mit Visual Studio .NET und den aktuellen Plattform-SDKs (siehe Soft-Link) liefert Microsoft einen C++-Compiler, der sich hauptsächlich der Portierbarkeit und der komfortablen Programmentwicklung widmet.

Intel plagt dagegen eine andere Sorge: Das Prozessorhaus beschleunigt laufend seine Prozessoren, doch die meiste Software lässt deren Möglichkeiten ungenutzt. Die Hyper-Threading-Technik der aktuellen Pentium-4-Prozessoren vergrößert die Diskrepanz zwischen Hard- und Software noch: Die meisten Programme besitzen nur einen einzigen Verarbeitungsstrang, und der zweite, virtuelle Prozessor wird nicht ausgelastet.

Intel will mit Version 7.0 seiner Compilerfamilie vor allem die Entwicklung von Multithreading-Programmen vereinfachen. Außerdem kamen Verbesserungen für Pentium 4 und den jungen 64-Bit-Itanium 2 hinzu. Neben dem C++-Compiler hat Intel auch den hauseigenen Fortran-Compiler aktualisiert und liefert beide sowohl für 32- und 64-Bit-Windows als auch für Linux [1]. Zum Ausprobieren stehen die Compiler im Internet bereit, für Linux gibt es bei privater Nutzung sogar eine dauerhafte Lizenz (siehe Soft-Link).

Dieser Artikel zeigt, was die Compiler für die Programmentwicklung unter 32-Bit-Windows bringen, und untersucht, wie gut

sie optimieren. Die Performance der Compiler ermittelten wir mit dem Integer-Teil der Benchmark-Suite CPU2000 der Standard Performance Evaluation Corporation (SPECint). Sie lief jeweils auf einem 2,8-GHz-Xeon (Hyper-Threading abgesteuert) und einem Athlon XP 1800+ (1,53 GHz).

Neuland

Die größten Änderungen an Microsofts Compiler sind auf die neue Umgebung .NET zurückzuführen. Damit C++-

An Prozessoren kennt Microsofts Compiler der siebten Generation jetzt auch Pentium II und III; mit dem Kommandozeilenparameter /G6 fasst er sie aber mit dem Pentium Pro zusammen und vom Pentium 4 mit seinen sehr unterschiedlichen Compiler-Anforderungen fehlt weiterhin jede Spur. Von den Vorversionen hat der Compiler die umfassenden Debugging-Fähigkeiten geerbt. Er unterstützt weiterhin Edit & Continue sowie das automatische Abdichten von Speicherlöchern [2]. Eine eigene Gruppe von Warnungen kümmert sich um Inkompatibilitäten zu 64-Bit-Windows. Visual Studio .NET schaltet diese per Default mit dem Schalter /Wp64 ein – eine Option, die der V7-Compiler von Intel nicht bietet.

Intel hat stark an der Zusammenarbeit seines Compilers mit Visual Studio .NET gearbeitet. Zwischen Microsoft- und Intel-Compiler wählt man nicht mehr wie bislang global für die ganze Entwicklungsumgebung, sondern gezielt in den Projekteigenschaften. Einzelne Quelldateien lassen sich fix mit einem der beiden Compiler verbinden. Bislang musste man das mit dem Präprozessor-Define USE_

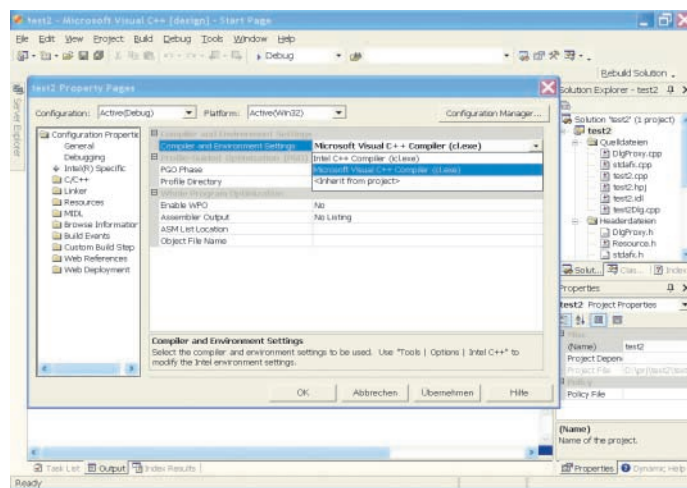
mit der deutschen Fassung von Visual Studio zurecht. Bei unseren Tests brachte er sie reproduzierbar beim Erstellen einer Anwendung zum Absturz, wohingegen es mit der englischen Visual-Studio-Version keinerlei Probleme gab. Wer 64-Bit-Programme entwickeln möchte, muss übrigens weiterhin Visual Studio 6.0 einsetzen.

Bei den Optimierungen geht Intel erheblich weiter als Microsoft. Natürlich kennt der Compiler Optimierungs-Flags für den Pentium 4 (/G7: Pentium-4-freundliche Instruktionsreihenfolge oder /qxW: mit Pentium-4-spezifischem Code wie SSE2). Der Compiler kann automatisch vektorisieren und damit manche Schleifen mit Hilfe der SSE- oder SSE2-Einheiten deutlich beschleunigen. Außerdem bietet er für Multiprozessor-beziehungsweise Hyper-Threading-Betrieb einen automatischen Parallelisierer. Dieser durchsucht Programme nach Stellen, die nebeneinander ablaufen können und verteilt diese selbstständig auf mehrere Threads.

Probalber übersetzen wir damit (Parameter /Qparallel) den Integer-Teil der Benchmark-Suite SPEC CPU2000, doch der Compiler fand in diesem Code nicht eine einzige lohnende Schleife. Genauso erging es uns mit einer aktuellen Fassung des c't-Apfelmännchens. Bei einer in C verfassten Variante des Linpack-1000-Benchmarks, senkten wir mit /Qpar_threshold1 den Schwellenwert für die Parallelisierung – und damit allerdings auch gleichzeitig die Performance. Die automatische Parallelisierung dürfte wohl nur in Einzelfällen bei sehr großen Projekten Früchte tragen.

Anders ist es bei der 'handgesteuerten' Parallelisierung mit Hilfe des anderen Intel-Compiler-Features OpenMP. Hier hat es der Programmierer in der Hand, gezielt das Multithreading für Optimierungen einzusetzen, ohne dass er sich mit der Erzeugung und Verwaltung von Threads abquälen muss.

Man programmiert intensive Berechnungen fast wie einen sequenziellen Programmablauf und gibt einem OpenMP-fähigen Compiler lediglich mittels Pragma-Zeilen Hinweise, welche Teile er wie parallelisieren kann. Eine einfache Zeile mehr



Gute Zusammenarbeit: Zwischen Intel- und Microsoft-Compiler wählt man jetzt in den Projekteigenschaften.

Programme damit klarkommen, erweiterte Microsoft die Sprache, damit die Software die .NET-Bibliotheken nutzen und man eigene .NET-Objekte erstellen kann. Der Compiler erzeugt dann keine nativ ausführbaren Programme, sondern Meta-Code in der Microsoft Intermediate Language (MSIL).

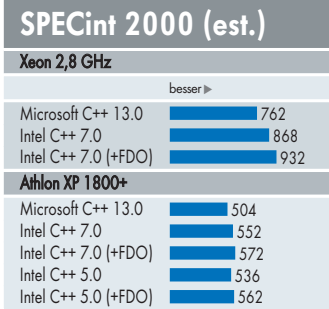
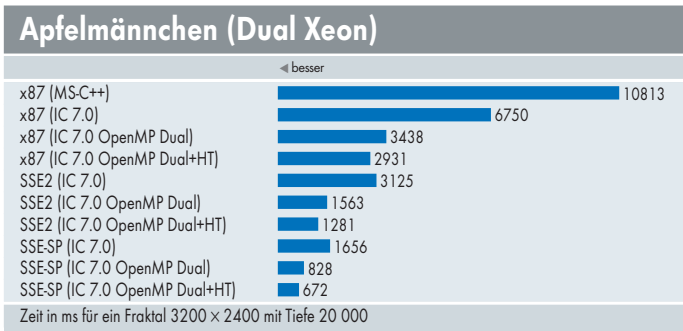
INTEL_COMPILER eintragen, jetzt kann man das genauso wie bei Intel-spezifischen Optimierungen bequem in den Projekteinstellungen neben jenen von Microsoft unter 'C/C++' vornehmen.

Wie bei früheren Compiler-Versionen von Intel kommt leider auch diese nicht auf Anhieb

```

1 #pragma omp parallel
2 {
3     unsigned short* buf; int y;
4     nid = omp_get_num_threads();
5     buf = (unsigned short*):AlignedAlloc(bufSize, 32);
6 #pragma omp for schedule(dynamic, 8) reduction(+: dwIterations)
7     for (y = 0; y < ymax; y++){
8         // hier die Berechnung
9         dwIterations += ...
10 #pragma omp critical
11     m_dwDone++; // or use InterlockedIncrement
12 } // for
13 ::AlignedFree(buf);
14 } // omp parallel
    
```

Mit OpenMP ist die Parallelisierung in Threads ein Kinderspiel.



macht etwa aus einer gewöhnlichen for-Schleife gleich mehrere Threads.

Dieser Ansatz bringt deutlich mehr als die automatische Parallelisierung. Das zeigt eine OpenMP-SSE/SSE2-Fassung des c't-Apfelmännchens, die uns freundlicherweise Intels Spezialist für schwierige Programmieraufgaben, Alex Klimovitski, mit nur drei OpenMP-Konstrukten erstellte. Auf einem Dual-Xeon-System erreicht man damit nahezu eine Verdoppelung der Performance. Und Hyper-Threading spielt ebenfalls gut 20 Prozent Performancegewinn ein, was schon erstaunlich ist, da der fraktale Algorithmus die internen Recheneinheiten stark auslastet und kaum Pausen durch Speicherzugriffe auftreten.

OpenMP

Bei der Gelegenheit kann eine in Assembler geschriebene Routine auch mal gleich das

Potenzial per Hand optimierter Routinen demonstrieren. Der Assemblercode für SSE2 ist noch einmal gut doppelt so schnell wie der C++-Code, und wenn man mit Single Precision zufrieden ist, kann die vierfache parallele Verarbeitung in der SSE-Einheit abermals eine Verdoppelung bewirken.

Microsofts C-Compiler ist in diesem Zusammenhang längst um Größenordnungen abgehängt. OpenMP und SSE2 kennt er nicht, und selbst bei der Einfachausführung ohne OpenMP bleibt der Code um 40 Prozent hinter Intels Kompilaten zurück.

Natürlich ist das Apfelmännchen die ideale Anwendung für OpenMP, da sich die Aufgabe leicht aufteilen lässt. Aber es zeigt das Potenzial auf und belegt, wie wenig Rechenzeit die automatische Verwaltung der Threads bei OpenMP benötigt.

Die restlichen Optimierungen verbesserte Intel gegenüber der Vorversion nur im Detail. Bei der Interprocedural-Optimisation (/Qipo) sortiert der Compiler den Programmablauf innerhalb einer Objektdatei neu und ersetzt kürzere Unterprogrammaufrufe durch Inline-Code. Diese Optimierung lässt sich auch auf verschiedene Objektdateien ausdehnen (/Qwp_ipo). In der Praxis weniger Bedeutung hat die profilgesteuerte Optimierung (PGO) mit zwei Compiler-Läufen. Nach dem ersten Com-

piler-Lauf verarbeitet die Anwendung Testdaten und generiert dabei Profilinformationen. Diese verwendet der Compiler, um im zweiten Durchgang spekulativ ausgeführte Programmzweige einzubauen. Diese Optimierung benötigt einen guten Testdatensatz, der viel Aufwand erfordert – ein schlechter macht das Programm eher größer und zugleich langsamer.

Profile

Für unseren Geschwindigkeitsvergleich ließen wir die Integer-Benchmarksuite der SPEC mit dem Intel-Compiler gleich zweimal laufen: Einmal mit und einmal ohne Profilloptimierung. Ansonsten verwendeten wir /O3, was auch den Vektorisierer einschaltet, und /Qipo. Den Prozessortyp gaben wir auf dem Xeon (nur ein Prozessor ohne HyperThreading) mit /QxW (Pentium 4, mit SSE/SSE2), auf dem Athlon XP mit /QxK (Pentium III, mit SSE) an.

Die Werte können allerdings nicht als Shootout zwischen den Prozessoren dienen (dazu ist der gewählte Athlon-XP zu langsam), sondern sie sollen zeigen, wie sich die beiden Compiler auf den wichtigsten Prozessorarchitekturen schlagen. So ist es wenig überraschend, dass der Intel-Compiler auf dem Xeon-System mit 932 SPECint(est.) an der Spitze liegt. Selbst ohne die Profilloptimierung erreicht er mit 868 zu 762 SPECint(est.) einen respektablen Abstand zum Microsoft-Compiler. Auf dem Athlon-XP 1800+ geht das Rennen zwischen den Compilern knapper aus: Mit 504 SPECint(est.) kommt Microsoft den 552 SPECint(est.) des Compiler-Konkurrenten Intel (ohne Profilloptimierung; mit 572) auf nur zehn Prozent Abstand nahe.

Microsofts Kombipaket für Pentium Pro bis III erzielt also auf dem Athlon XP eine

brauchbare Geschwindigkeit. Die Fähigkeiten des Pentium 4 lässt es aber brachliegen, hier zeigt der Intel-Compiler seine beträchtlichen Stärken.

Naturgemäß verwendet auch AMD den Intel-Compiler für die offiziellen SPEC-Messungen (eigene 32-Bit-Compiler hat AMD ja nicht zu bieten). Als dessen Version 6 erschien, hielt AMD allerdings an dem Fünfer-Vorgänger fest, da er auf Athlon-Systemen bessere Ergebnisse erzielte. Probalber haben wir auf dem Athlon XP auch mit Version 5 des Intel-Compilers gemessen. Sie bringt bei den gleichen Optimierungen aber mit 536 bzw. 562 SPECint(est.) im Schnitt weniger als die neue Version (552 und 572 SPECint(est.)), sodass AMD getrost wieder – Intel sei Dank – mit aktuellen Compilern arbeiten kann.

Bei den Übersetzungszeiten liegt allerdings Microsofts Compiler weit vorn, nur 95 s benötigt er für die komplette SPEC-Suite auf dem Xeon und 153 s dauert es auf dem Athlon XP. Intels Compiler genehmigt sich dabei deutlich mehr Zeit: 292 s auf dem Xeon und 626 s auf dem Athlon XP.

Fazit

Beim Entwickeln und Entwerfen von Software kann Intel dem Duett aus Visual Studio und Microsofts Compiler nur schwer das Wasser reichen. Dazu reichen schon die deutlich längeren Compile-Zeiten und das Fehlen von Edit & Continue. Ist das Programm aber einmal fertig, gibt es nirgends zehn bis 15 Prozent mehr Leistung so günstig: Einfach das Programm neu übersetzen, fertig. Mit OpenMP bietet Intels Compiler zusätzlich eine deutliche Vereinfachung der Parallel-Programmierung. Letztlich wird man daher oft beide nebeneinander nutzen; nur besser Deutsch lernen sollte man bei Intel noch. (as)

Literatur

[1] Linux-Sprinter, Intel C++-Compiler für Linux, c't 23/01, S. 222
 [2] Innere Sicherheit, Laufzeitprüfungen von Visual C++ Version 7, Matthias Withopf, c't 10/02, S. 222

Compiler-Preise	
Microsoft	
C++ .NET Standard	109 US-\$
Visual Studio .NET Profess.	1079 US-\$
Upgrade	549 US-\$
Intel	
C/C++ 7.0 Windows	399 US-\$
Academic Licence	100 US-\$
Upgrade	kostenlos von V5.0 oder V6

