EINE KURZE EINFÜHRUNG ZUR VERWENDUNG VON THREADS IN GAUSS

Der Begriff Thread stammt von dem Ausdruck "Thread of Execution" (Ausführungsstrang) – er bezeichnet einfach einen Abschnitt des Codes, der ausgeführt werden soll. Ein Single-Thread-Programm verfügt nur über einen einzelnen Ausführungsstrang, d.h. das Programm selbst – auch bekannt als „Hauptthread“. Im Gegensatz dazu kann ein Multi-Thread-Programm mehrere Threads – Codeabschnitte – enthalten, die *gleichzeitig* ausgeführt werden. Da diese Threads Teil desselben Programms sind, teilen sie sich denselben Arbeitsbereich, d.h., sie arbeiten mit denselben Symbolen.

Damit verfügen Sie bereits über die einzig wirklich wichtige Information im Bezug auf Threads: Threads sind separate Abschnitte des gleichen Programms, die gleichzeitig ausgeführt werden und mit den gleichen Daten arbeiten. Diese Information ist tatsächlich so wichtig, dass sie gleich noch mal in etwas anderen Worten wiederholt werden soll:

Threads sind separate Abschnitte in einem Programm, die zur gleichen Zeit laufen, denselben Arbeitsbereich verwenden und beim Abarbeiten auf die gleichen Symbole zugreifen.

Der herausragende Vorteil beim Multithreading ist der erhebliche Zeitgewinn, der durch die Nutzung von Multicore- und Multiprozessorsystemen erzielt wird.

Dies wirft Fragen auf zum Workflow und zur Datenintegrität. Wie werden die Erstellung und Ausführung von Threads verwaltet und die von ihnen geleistete Arbeit genutzt? Und wie wird die Datenintegrität bewahrt? (Sie möchten *nicht*, dass zwei Threads gleichzeitig demselben Symbol zugewiesen sind.)

Um den Thread-Workflow abzuarbeiten, verwendet GAUSS 9.0 den Ansatz "Split & Merge" (*Teilen und Zusammenfügen*) bzw. "Create & Wait" (*Erzeuge und Warte*). An verschiedenen Punkten in Ihrem Programm erstellen Sie eine Definition der Threads, die dann erzeugt und als Gruppe ausgeführt wird. Sobald sie erstellt wurde, werden die Threads dieser Definition gleichzeitig ausgeführt. Der Hauptthread wartet darauf, dass die erstellten Threads abgeschlossen werden, und verwendet dann in seinem weiteren Arbeitsverlauf ihre Ergebnisse.

Um die Datenintegrität zu bewahren, führen wir die Programmierungsregel "jeder Thread kann lesen, es sei denn, ein Thread schreibt" ein (formell ausgedrückt, der 'writer-must-isolate' - '*Schreiber-muss-isolieren*'). Das bedeutet, dass so viele Threads wie gewünscht in einer Definition auf Symbole hinweisen können. Voraussetzung dafür ist ausschließlich, dass aus ihnen nur gelesen wird, ihnen aber nichts zugewiesen wurde. Symbole, für die eine Zuweisung erfolgt ist, müssen einem einzelnen Thread jedoch *vollständig* zur Verfügung stehen. Kein anderer Thread der Definition kann auf dieses Symbol hinweisen. Sie können ihm weder zugewiesen werden, noch aus ihm lesen. Sie können also überhaupt keinen Bezug auf das Symbol nehmen.

Beachten Sie, dass diese Einschränkung sich nur auf Threads innerhalb einer gegebenen Definition bezieht (einschließlich aller Definitionen von Unterthreads, die sie möglicherweise erstellt haben). Sie wird nicht auf Thread-Definitionen angewendet, die unmöglich gleichzeitig laufen können.

Für Threads, die im Hauptcode definiert wurden, bezieht sich die 'writer-must-isolate'-Regel auf die allgemeinen Symbole. Für Threads, die in PROCs oder KEYWORDs definiert wurden, bezieht sie sich auf die allgemeinen Symbole, lokalen Symbole und die PROC / KEYWORD-Argumente.

DIE FUNKTIONEN

GAUSS 9.0 führt vier Schlüsselwörter für das Multithreading Ihres Programms ein:

```
ThreadStat
ThreadBegin
ThreadEnd
ThreadJoin
```

`ThreadStat` definiert eine einzelne Anweisung, die als ein Thread ausgeführt werden soll:

```
ThreadStat n = m'm;
```

`ThreadBegin` und `ThreadEnd` definieren einen mehrzeiligen Code-Block, der als ein Thread ausgeführt werden soll:

```
ThreadBegin;
  y = x'x;
  z = y'y;
ThreadEnd;
```

Zusammen erstellen diese Schlüsselwörter "Definitionen" von Threads, die gleichzeitig ausgeführt werden sollen:

```
ThreadStat n = m'm;
ThreadBegin;
  y = x'x;
  z = y'y;
ThreadEnd;
ThreadBegin;
  q = r'r;
  r = q'q;
ThreadEnd;
ThreadStat p = o'o;
```

Schließlich beendet `ThreadJoin` die Definition der verschiedenen Threads. Es wartet darauf, dass die Threads in einer Definition abgeschlossen werden und wieder zum Hauptthread zurückkehren, der dann fortfahren kann:

```
ThreadBegin;
  y = x'x;
  z = y'y;
ThreadEnd;
ThreadBegin;
  q = r'r;
  r = q'q;
ThreadEnd;
ThreadStat n = m'm;
ThreadStat p = o'o;
ThreadJoin;
b = z + r + n'p;
```

WAS MIT THREADS MÖGLICH IST

Es gibt zwei Hauptaspekte hinsichtlich des Programmierens mit Threads.

1. Sie können Threads überall erstellen. Sie können sie im Hauptcode erzeugen, in PROCs, in KEYWORDs und sogar innerhalb anderer Threads.

2. Sie können PROCs und KEYWORDs von Threads aus aufrufen. Letztendlich ist es diese Option, die wirklich alles miteinander verbindet. Sie können ein PROC aus einem Thread aufrufen, dieser PROC wiederum kann Threads erstellen, jeder dieser Threads kann dann wieder PROCs aufrufen, von denen jeder wieder Threads erstellen kann und so weiter und so fort.

Sie können also z.B. Folgendes tun:

```
q = chol(b);
ThreadBegin;
   x = q + m;
   ThreadBegin;
      y = x'x;
      z = q'm;
   ThreadEnd;
   ThreadBegin;
      a = b + x;
      c = a + m;
   ThreadEnd;
   ThreadJoin;
   q = m'c;
ThreadEnd;
```

```
ThreadBegin;
   ThreadStat r = m'm;
   ThreadStat s = m + inv(b);
   ThreadJoin;
   t = r's;
ThreadEnd;
ThreadJoin;
x = r+s+q+z-t;
```

Noch wichtiger – es ist möglich, folgende Definition auszuführen:

```
proc bef(x);
   local y,t;

   ThreadStat y = nof(x);
   ThreadStat t = dof(x'x);
   ThreadJoin;
   t = t+y;

   retp(t);
endp;

proc abr(m);
   local x,y,z,a,b;

   a = m'm;
   ThreadStat x = inv(m);
   ThreadStat y = bef(m);
   ThreadStat z = dne(a);
   ThreadJoin;
   b = chut(x,y,z,a);

   retp(inv(b));
endp;

s = rndn(500,500);
ThreadStat t = abr(s);
ThreadStat q = abr(s^2);
ThreadStat r = che(s);
ThreadJoin;
w = del(t,q,r);
print w[1:10,1:10];
```

Sie können also alles, was Sie wollen, mit Multithreading bearbeiten und es überall aufrufen. Es ist möglich, alle PROCs und KEYWORDs in Ihren Bibliotheken mit Multithreading zu bearbeiten und sie beliebig in Ihren Multithread-Programmen aufzurufen.

EINSCHRÄNKUNGEN

Ein paar Punkte sind im Bezug auf Kodierungseinschränkungen noch zu beachten. Zum einen können Sie keine Thread-Definitionsanweisungen und reguläre Anweisungen miteinander verknüpfen. Folgendes ist nicht möglich:

```
ThreadStat a = b'b;
n = q;
ThreadStat c = d'd;
ThreadJoin;
```

Oder dies:

```
if k == 1;
   ThreadStat a = b'b;
elseif k == 2;
   ThreadStat a = c'c;
endif;
if j == 1;
   ThreadStat d = e'e;
elseif j == 2;
   ThreadStat d = f'f;
endif;
```

```
ThreadJoin;
```

Jede Definition von Threads wird als eine Gruppe erstellt und immer von einem `ThreadJoin` abgeschlossen, wie hier zu sehen ist:

```
n = q;
ThreadStat a = b'b;
ThreadStat c = d'd;
ThreadJoin;
```

Und hier:

```
ThreadBegin;
   if k == 1;
      a = b'b;
   elseif k == 2;
      a = c'c;
   endif;
ThreadEnd;
ThreadBegin;
   if j == 1;
      d = e'e;
   elseif j == 2;
      d = f'f;
   endif;
ThreadEnd;
ThreadJoin;
```

Zum anderen können Sie zwar, wie in der Übersicht bereits erwähnt, in so vielen Threads, wie Sie wollen, auf „Nur Lesen“-Symbole innerhalb einer Definition hinweisen, aber alle Symbole, denen etwas zugewiesen wurde, müssen einem einzelnen Thread *vollständig* zur Verfügung stehen. Ein Symbol, dem ein Thread zugewiesen wurde, kann nicht von einem anderen Thread in dieser Definition geschrieben *oder gelesen* werden. Das ist die 'writer-must-isolate'-Regel.

Demzufolge können Sie Folgendes ausführen:

```
ThreadStat x = y'y;
ThreadStat z = y+y;
ThreadStat a = b-y;
ThreadJoin;
```

Dies wiederum ist nicht möglich:

```
ThreadStat x = y'y;
ThreadStat z = x'x;
ThreadStat a = b-y;
ThreadJoin;
```

Der Grund dafür ist, dass die Threads innerhalb einer Definition gleichzeitig laufen. Daher besteht keine Möglichkeit zu wissen, wann eine Zuweisung zu einem Symbol stattgefunden hat, oder von einem Thread aus in Erfahrung zu bringen, wie der „Status“ eines Symbols in einem anderen Thread ist.

Im folgenden Beispiel soll noch mal kurz auf den verschachtelten Thread eingegangen werden, um darzustellen, inwiefern die 'writer-must-isolate'-Regel auf ihn angewendet wird:

```
q = chol(b);          // main code, no threads yet
ThreadBegin;          // Th1: isolates x,y,z,a,c,q from Th2
   x = q + m;
   ThreadBegin;          // Th1.1: isolates y,z from 1.2
      y = x'x;
      z = q'm;
   ThreadEnd;
   ThreadBegin;          // Th1.2: isolates a,c from 1.1
      a = b + x;
      c = a + m;
   ThreadEnd;
   ThreadJoin;           // Joins 1.1, 1.2
   q = m'c;
```

```
   ThreadEnd;
   ThreadBegin;          // Th2: isolates r,s,t from Th1
      ThreadStat r = m'm;   // Th2.1: isolates r from 2.2
      ThreadStat            // Th2.2: isolates s from 2.1
         s = m + inv(b);
      ThreadJoin;           // Joins 2.1, 2.1
      t = r's;
   ThreadEnd;
   ThreadJoin;           // Joins Th1, Th2
   x = r+s+q+z-t;
```

Das Wichtigste hierbei ist, dass alle Symbole, denen ein Thread *oder seine Unterthreads* zugewiesen sind, von allen anderen Threads (und seinen Unterthreads) derselben Verschachtelungsebene in dieser Definition isoliert sein müssen. Auf der anderen Seite können Unterthreads eines Threads beliebig Symbole lesen/schreiben, die von ihrem Hauptthread gelesen/geschrieben wurden, da hier keine Gefahr der Gleichzeitigkeit besteht; sie müssen lediglich geschriebene Symbole von ihren Siblings (Geschwister) und Unterthreads dieser Siblings isolieren.

Wenn Sie die 'writer-must-isolate'-Regel nicht einhalten, stürzt Ihr Programm ab. Schlimmer ist allerdings, dass es zuvor unbestimmte Ergebnisse erzeugt.

Am Ende ist `ThreadEnd` der Befehl, der einem Thread sagt, dass er beendet wird. Deshalb dürfen Sie keinen Code schreiben, der einen Thread daran hindert, diesen Befehl zu erreichen. Geben Sie z.B. RETP nicht in der Mitte eines Threads ein:

```
   ThreadStat m = imt( 9 );
   ThreadBegin;
      x = q[1];
      if x = 1;
         retp(z);
      else;
         r = z + 2;
      endif;
   ThreadEnd;
   ThreadJoin;
```

Verwenden Sie kein GOTO, um in die oder aus der Mitte eines Threads zu springen:

```
retry:
   ThreadBegin;
      { err, x } = fna(q);
      if err;
         goto badidea;
      endif;
      x = fnb(x);
   ThreadEnd;
   ThreadStat y = fnb(y);
   ThreadJoin;
   z = fnc(x,y);
   save z;
   end;

badidea:
   errorlog "Error computing fna(q)";
   q = fnd(q);
   goto retry;
```

Tun Sie grundsätzlich nichts, was einen Thread davon abhalten kann, den Befehl `ThreadEnd` zu erreichen. Nur durch ihn weiß er, dass seine Aufgabe erfüllt ist. Es ist jedoch kein Problem, END und STOP aufzurufen. Sie beenden das Programm wie üblich und schließen alle laufenden Threads in dem Prozess ab.

(Sie *können* GOTO und Labels verwenden, um innerhalb eines Threads hin und her zu springen, genauer gesagt innerhalb der Grenzen eines `ThreadBegin/ThreadEnd`-Paares.)