Parallelisierung

Die Funktion `calculate` wurde in zwei Methoden aufgeteilt, die aufgerufen werden, je nach dem, ob das Jacobi- oder das Gauß-Seidel-Verfahren ausgewählt wurde:

```
static void calculate (...)
{
  if (options->method == METH_JACOBI)
  {
    calculate_jacobi(arguments, results,
                     options);
  }
  else
  {
    calculate_gauss_seidel(arguments, results,
                           options);
  }
}
```

`calculate_gauss_seidel` entspricht im wesentlichen dem Original.
`calculate_jacobi` ruft intern die Funktion `calculate_jacobi_fun` auf:

```
static void calculate_jacobi (...)
{
  ...
  int temp;
  pthread_t threads[options->number];
  pthread_mutex_t mutex;
  ...
  assert(!pthread_mutex_init(&mutex, NULL));
  unsigned int cells_per_thread = (N - 1)
                                  / options->number;
  struct jacobi_fun_args args[options->number];

  while (term_iteration > 0)
  {
    double** Matrix_Out = arguments->Matrix[m1];
    double** Matrix_In  = arguments->Matrix[m2];

    maxresiduum = 0;
```

```
    for(unsigned int k = 0; k < options->number; k++){
      args[k].first_i = k * cells_per_thread + 1;
      args[k].last_i = (k == options->number) ? N - 1
                       : (k + 1) * cells_per_thread;
      args[k].maxresiduum = &maxresiduum;
      args[k].residuummutex = &mutex;
      args[k].N = N;
      args[k].opts = options;
      args[k].fpisin = fpisin;
      args[k].pih = pih;
      args[k].Matrix_In = Matrix_In;
      args[k].Matrix_Out = Matrix_Out;
      args[k].term_iteration = term_iteration;
      assert(!pthread_create(&threads[k], NULL,
             &calculate_jacobi_fun, &args[k]));
    }

    for(unsigned int k = 0; k < options->number; k++){
      pthread_join(threads[k], NULL);
    }

    results->stat_iteration++;
    results->stat_precision = maxresiduum;
    ...
}
```

Hierbei wurde `maxresiduum` als Pointer übergeben und jeweils per Mutex gelockt, falls darin geschrieben werden sollte. An `calculate_jacobi_fun` wird jeweils ein `first_i` und ein `last_i` übergeben, darin wird über die übergebenen Zeilen iteriert:

```
void* calculate_jacobi_fun(void* _args)
{
  unsigned int i, j;
  double star;
  double residuum;

  struct jacobi_fun_args *args =
                       (struct jacobi_fun_args*)_args;

  /* over all rows */
  for (i = args->first_i; i <= args->last_i; i++)
```

```
  {
    ...
  }
  return NULL;
}
```

Messung

Messdaten

#Threads	Laufzeit in s bei Lauf n:				Durchschnittlicher Speedup
	1	2	3	Durchschnitt	
Original	571,92	572,06	572,56	572,18	–
1	622,22	621,99	622,19	622,13	1,00
2	317,25	317,21	317,29	317,25	1,96
3	216,74	216,79	216,72	216,75	2,87
4	167,38	167,87	168,46	167,90	3,71
5	140,61	140,77	140,46	140,61	4,42
6	124,02	124,84	124,01	124,29	5,01
7	114,33	115,42	114,61	114,79	5,42
8	106,41	107,12	107,73	107,09	5,81
9	98,52	100,46	99,76	99,58	6,25
10	89,63	91,76	89,55	90,31	6,89
11	82,91	81,07	78,85	80,94	7,69
12	75,03	74,60	74,75	74,79	8,32

Diskussion

Offenbar ist unsere parallelisierte Variante mit nur einem Thread langsamer als `partdiff-seq`. Möglicherweise entsteht an einer Stelle sehr viel Overhead durch die Threads oder Teile des Codes wurden nicht ausreichend optimiert. Beispielsweise wird in `calculate_jacobi` bei jeder Iteration ein neues Array `args` erstellt, obwohl sich der Inhalt eigentlich nicht ändert. Der Compiler hat das offenbar trotz `Ofast` nicht optimiert. Möglicherweise verbraucht auch das Locking per Mutex, welches bei einem einzelnen Thread überflüssig ist, aber stets vorgenommen wird, sehr viel Zeit. Insgesamt hätte man noch viel Performance herausholen können, aber dafür war leider zu wenig Zeit.
Zumindest läuft das Programm mit 12 Threads um den Faktor 8,32 schneller.

Es ergibt sich folgendes Speedup-Diagramm: