OpenMP

Module 5

INF8601 Systèmes informatiques parallèles

Michel Dagenais

École Polytechnique de Montréal
Département de génie informatique et génie logiciel

Sommaire

OpenMP

OpenMP

- Programmation en mémoire partagée, (Open Multi-Processing, OpenMP).
- API C, C++, Fortran.
- Sur Linux, Unix, AIX, Solaris, Mac OS X, et Microsoft Windows.
- Consortium sans but lucratif qui inclut AMD, IBM, Intel, Cray, HP, Fujitsu, NVIDIA, NEC, Microsoft, Texas Instruments, VMware, Oracle Corporation...

Versions

- 1997, OpenMP 1.0 pour Fortran (C/C++ en 1998)
- 2000, OpenMP 2.0 pour Fortran (C/C++ en 2002)
- 2005, OpenMP 2.5 pour C/C++ et Fortran
- 2008, OpenMP 3.0
- 2011, OpenMP 3.1
- 2013, OpenMP 4.0 avec SIMD
- 2015, OpenMP 4.5 avec taskloop et ajustements
- 2018, OpenMP 5.0 avec loop, boucles plus flexibles
- 2020, OpenMP 5.1 avec scope et assume

Caractéristiques

- Standard complet et portable, disponible gratuitement.
- Utilisation de pragmas pour déclarer les sections à paralléliser, à synchroniser...(utilisé par le compilateur ou le pré-compilateur).
- Librairie de support à l'exécution, variables d'environnement.
- Support pour le traçage et le débogage.
- Parallélisme à grain fin qui peut exploiter les coeurs du CPU et le GPU.
- Axé sur la performance.

Alternatives

- Compilateur parallélisant.
- Intel Threading Building Blocks.
- OpenMP permet de paralléliser un programme séquentiel graduellement avec une granularité très fine, et même à partir de 4.0 d'utiliser le GPU.

Modèle

- Possibilité de créer un grand nombre de fils d'exécution, plus grand ou égal au nombre des processeurs (omp parallel), et de diviser le travail entre les fils (omp for, sections, tasks...).
- Tous les fils d'une région parallèle doivent passer par les mêmes sections de division du travail, même s'ils peuvent être dans des fonctions appelées.
- La synchronization de contrôle découle en bonne partie des pragmas insérés (section parallèles ou séquentielles, barrières).
- Variables peuvent être privées (copie locale à chaque fil) ou partagées.
- La synchronisation des données partagées doit se faire par verrous.

OpenMP

Les pragmas

- Syntaxe: `#pragma omp directive [clause...]`
- En C++ 11 ce peut être:
  `[[ omp :: directive( directive-name, clause...) ]]`
- Directives: parallel, for, sections, single, parallel for, parallel sections, simd, target, declare variant, dispatch, requires, assume, nothing, error, teams, masked, loop, tile, unroll, cancel, cancellation point.
- Clauses: shared, private, firstprivate, lastprivate, default, reduction, copyin, if, order, ordered, schedule, nowait, safelen, linear, aligned, collapse, device, map, num_threads, proc_bind, ...

Parallel

- Des fils d'exécution s'ajoutent au fil principal au début du bloc pour former une équipe (team) de fils.
- Le fil principal attend que tous les autres fils aient fini à la fin du bloc.
- Chaque fil d'exécution fait la même chose.
- Apporter des variantes en utilisant son numéro de fil d'exécution (omp_get_thread_num()).

Exemple

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

int main( int argc, char **argv ) {
  int rank, size;
  #pragma omp parallel private(rank)
  { rank= omp_get_thread_num();
    size= omp_get_num_threads();
    printf( "Hello world! I'm %d of %d\n",rank, size );
  }
  return 0;
}
dorsal> export OMP_NUM_THREADS=4
dorsal> gcc -fopenmp -o HelloWorld HelloWorld.c
dorsal> ./HelloWorld
Hello world! I'm 0 of 4
Hello world! I'm 2 of 4
Hello world! I'm 3 of 4
Hello world! I'm 1 of 4
```

Teams

- Des équipes (team) de fils d'exécution s'ajoutent à l'équipe principale au début du bloc pour former une ligue (league) d'équipes de fils.
- Typiquement utilisé pour les processeurs multiples, chacun avec de nombreux fils d'exécution, sur GPU.
- Chaque équipe fait la même chose.
- Apporter des variantes en utilisant son numéro d'équipe (omp_get_team_num()).

Exemple

```
float sp_x[N], sp_y[N], sp_a=0.0001e0;
double dp_x[N], dp_y[N], dp_a=0.0001e0;

#pragma omp teams num_teams(2) private(tm_id)
{ tm_id = omp_get_team_num();
  if(tm_id == 0) // Do Single Precision Work (SAXPY) {
    #pragma omp parallel for simd simdlen(8)
    for(int i=0; i<N; i++){sp_x[i] = sp_a*sp_x[i] + sp_y[i];}
  }
  if(tm_id == 1) // Do Double Precision Work (DAXPY) {
    #pragma omp parallel for simd simdlen(4)
    for(int i=0; i<N; i++){dp_x[i] = dp_a*dp_x[i] + dp_y[i];}
  }
}
printf("i=%d sp|dp %f %f \n",N-1, sp_x[N-1], dp_x[N-1]);
printf("i=%d sp|dp %f %f \n",N/2, sp_x[N/2], dp_x[N/2]);
```

Sections

- Diviser une séquence de code en sections, chacune n'est exécutée que par un seul fil d'exécution.

```
#pragma omp parallel
{ #pragma omp sections
  { #pragma omp section
    { ...
    }
    #pragma omp section
    { ...
    }
  }
  ...
}
```

Exemple

```
#pragma omp parallel sections
{ #pragma omp section
  { for (int i=0; i<n; i++) {
      input(i);
      signal_input(i);
    }
  }
  #pragma omp section
  { for (int i=0; i<n; i++) {
      wait_input(i);
      process(i);
      signal_process(i);
    }
  }
  #pragma omp section
  { for (int i=0; i<n; i++) {
      wait_process(i);
      output(i);
    }
  }
}
```

For

- Le for doit avoir une forme simple, sans `break` ou `continue`.
- le domaine de l'itérateur est divisé entre les fils pour tout couvrir.
- Différentes stratégies de division du travail sont possibles (schedule: statique, dynamique, guidée, auto, runtime).
- Chaque fil d'exécution s'occupe d'un intevalle de la boucle.
- Il est possible d'avoir un combiné `#pragma omp parallel for`.

Exemple

Le fil 0 pourrait faire i = 0, 1, 2, 3, 4 et le fil 1 pourrait faire i = 5, 6, 7, 8, 9

```
#pragma omp parallel for default(none)
    private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++) {
  sum = 0.0;
  for (j=0; j<n; j++) sum += b[i][j]*c[j];
  a[i] = sum;
}
```

Transformation de boucle

```
#pragma omp tile sizes(size-list)
loop-nest

#pragma omp unroll [clause]
loop-nest
```

- Le compilateur peut changer la structure des boucles visées.
- Unroll déroule les boucles pour réduire le surcoût inhérent à chaque tour de boucle (incrément d'indice, comparaison, saut).
- Tile fait des groupements multi-dimensionnels d'indices et commence par itérer entre les groupements et ensuite itère à l'intérieur des groupements. Cela double la profondeur d'imbrication.

Scan

```
{
structured-block-sequence
#pragma omp scan
structured-block-sequence
}
```

- Calcul préfixe inclusif ou exclusif pour la boucle englobante.
- Un premier bloc constitue la première phase et le second bloc constitue la seconde phase du calcul préfixe.

Exemple

```
int a[N], b[N];
int x = 0;

for (int k = 0; k < N; k++) a[k] = k + 1;

#pragma omp parallel for simd reduction(inscan,+: x)
for (int k = 0; k < N; k++) {
  x += a[k];
  #pragma omp scan inclusive(x)
  b[k] = x;
}

printf("x = %d, b[0:3] = %d %d %d\n", x, b[0], b[1], b[2]);
```

Tasks

- Diviser une séquence de code en tâches, chacune n'est exécutée que par un seul fil d'exécution, parmi ceux de l'équipe de la section parallèle englobante, en différé.

```
#pragma omp parallel
{ #pragma omp task
  { ...
  }
  #pragma omp task
  { ...
  }
  ...
}
```

Exemple

```
#pragma omp parallel
{ #pragma omp single
  { #pragma omp task
    { printf("Hello\n"); }

    #pragma omp task
    { printf("World\n"); }

    printf(“Bye\n“);
  }
}
```

Single

- Cette section de code n'est exécutée que par un seul fil d'exécution.

```
#pragma omp parallel
{ #pragma omp single
  { ...
  }
  ...
}
```

Exemple

```
#pragma omp parallel
{ #pragma omp single
  printf("Beginning work1.\n");
  work1();
  #pragma omp single
  printf("Finishing work1.\n");
  #pragma omp single nowait
  printf("Finished work1 and beginning work2.\n");
  work2();
}
```

Masked

- Cette section de code n'est exécutée que par les fils d'exécution spécifiés dans la clause *filter*. Par défaut, sans clause filter, seul le fil d'exécution 0 est pris.

```
#pragma omp parallel
{ #pragma omp masked
  { ... }
  ...
}
```

Exemple

```
#pragma omp parallel
{ do
  { #pragma omp for private(i)
    for( i = 1; i < n-1; ++i )
    { xold[i] = x[i]; }

    #pragma omp single
    { toobig = 0; }

    #pragma omp for private(i,y,error) reduction(+:toobig)
    for( i = 1; i < n-1; ++i ) {
      y = x[i];
      x[i] = average(xold[i-1], x[i], xold[i+1]);
      error = y - x[i];
      if(error > tol || error < -tol) ++toobig;
    }
    #pragma omp masked
    { ++c;
      printf( "iteration %d, toobig=%d\n", c, toobig );
    }
  } while( toobig > 0 );
}
```

Scope

- Cette section de code est exécutée par tous les fils d'exécution mais en tenant compte des clauses *private*, *reduction*, *nowait* spécifiées.

Critical

- `#pragma omp critical [name]`: une seule région critique du même nom peut s'exécuter en même temps, même par des fils d'autres régions parallèles.
- Constitue un verrou global ou partiel (avec un nom) plutôt qu'un verrou associé à un élément de donnée particulier.

Exemple

```
#pragma omp parallel shared(x, y) private(ix_next, iy_next)
{ #pragma omp critical (xaxis)
  { ix_next = dequeue(x); }

  work(ix_next, x);

  #pragma omp critical (yaxis)
  { iy_next = dequeue(y); }

  work(iy_next, y);
}
```

Synchronisation de contrôle

- Lorsque des données peuvent être partagées par des fils d'exécution différents et qu'il y a une dépendance (lecture d'une donnée écrite précédemment), il faut mettre une barrière pour synchroniser les fils parallèles.
- Barrière implicite à la fin de chaque région.

Exemple

```
x = 2;
#pragma omp parallel num_threads(2) shared(x)
{ if (omp_get_thread_num() == 0) {
    x = 5;
  } else {
    /* Print 1: the following read of x has a race */
    printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x );
  }

  #pragma omp barrier

  if (omp_get_thread_num() == 0) {
    printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
  } else {
    printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
  }
}
```

Taskwait

- `#pragma omp taskwait`
- Attendre après toutes les tâches enfant de la tâche courante.

Exemple

```
#pragma omp parallel
{ #pragma omp single
  { #pragma omp task
    { printf("Hello\n"); }

    #pragma omp task
    { printf("World\n"); }

    #pragma omp taskwait
    printf(“Bye\n“);
  }
}
```

Atomic

- `#pragma omp atomic {x <operateur>= expression}`:
- Demander une opération arithmétique atomique sur la variable accédée.

Exemple

```
for (i = 0; i < 10000; i++) {
  index[i] = i % 1000;
  y[i]=0.0;
}

for (i = 0; i < 1000; i++) {
  x[i] = 0.0;
}

#pragma omp parallel for shared(x, y, index, n)
for (i=0; i<n; i++) {
  #pragma omp atomic update
  x[index[i]] += work1(i);

  y[i] += work2(i);
}
```

Flush

- `#pragma omp flush`
- Barrière mémoire complète. La vue temporaire de la mémoire par un fil est synchronisée avec la copie principale
- Doit être utilisé dans les deux ou plusieurs fils d'exécution qui accèdent des données partagées en parallèle.

Exemple

```
// Producteur

data = 2000;
#pragma omp flush(data);
flag = 1;
#pragma omp flush(flag);

// Consommateur
#pragma omp flush(flag)
if(flag == 1) {
  #pragma omp flush(data)
  x = data;
}
```

Ordered

- Cette section de code est exécutée dans l'ordre des itérations de la boucle qui la contient.
- Très contraignant pour l'exécution parallèle de la boucle.

```
#pragma omp parallel for ordered
{ for(i = 0 ; i < N; i++) {
    calcul complexe...

    #pragma omp ordered
    { printf("a[%d]=%d", i, a[i]);
    }
  }
}
```

Dépendances explicites

- `#pragma omp depobj(obj)`, déclarer obj comme objet pour représenter une dépendance et changer son état.
- Clause `depend`, spécifier une dépendance et son type pour une tâche (task) ou pour une exécution sur une cible (target).

Exemple

```
int a = 1, b = 2, c;

#pragma omp parallel
  #pragma omp single
  {
    #pragma omp task depend(out:a)
    { a++; printf ("Task 1; "); }

    #pragma omp task depend(out:b)
    { b++; printf ("Task 2; "); }

    #pragma omp task depend(in:a,b) depend(out:c)
    { c = a + b; printf ("Task 3; "); }

    #pragma omp task depend(in:c)
    { printf ("Task 4: c = %d;\n", c); }

Task 2; Task 1; Task 3; Task 4: c = 5;
Task 1; Task 2; Task 3; Task 4: c = 5;
```

OpenMP

Fonctions de synchronisation

- omp_init_lock, omp_destroy_lock, omp_set_lock, omp_unset_lock, omp_test_lock: verrous pour une granularité de synchronisation plus fine.
- Aussi des verrous qui peuvent être imbriqués (nest_lock).

Statut des variables

- Pour chaque région parallèle ou division du travail (parallel, task, for, sections), il faut définir la portée des variables et comment elles sont initialisées et finalisées.
- Variables locales à la région, locales au fil d'exécution ou partagées. Une clause permet de spécifier ce qui est par défaut.

Shared, Private

- La variable visée doit exister avant et après la région visée par la directive et former un objet complet en mémoire.
- L'effet est sur la portée statique et non dynamique de la directive.
- Shared, une seule copie partagée par tous les fils d'exécution.
- Private: variable privée à chaque fil d'exécution, indéfinie avant et après la région visée par la directive.
- Firstprivate: variables privées initialisées avec la valeur en entrée.
- Lastprivate: la valeur de la variable privée du fil qui effectue la dernière itération de la boucle est prise comme valeur en sortie.
- Copyprivate: propager une variable (private ou threadprivate) d'une région `single` aux autres fils d'exécution.

Threadprivate

- Une copie des variables listées est créée pour chaque fil d'exécution.
- La clause copyin permet d'effectuer de nouveau la copie plus tard.
- Ces variables copiées continuent d'exister d'une tâche à l'autre et possiblement d'une section parallèle à l'autre.

Reduction

- Variable utilisée pour une réduction avec un opérateur commutatif et associatif entre les fils d'une région parallèle.
- `reduction(+: sum)`

If

- La région visée n'est effectuée en parallèle que si la condition est vraie.
- Dans plusieurs cas il est préférable de rester en séquentiel si le nombre de processeurs ou la taille du problème sont trop petits.

Nowait

- Lorsque les données sont indépendantes, l'exécution de deux régions peut se chevaucher.

```
#pragma omp parallel shared(n,a,b,c,d,e,f) private(i,scale) {
  #pragma omp for nowait
  for (i=0; i<n; i++) a[i] = b[i] + c[i];

  #pragma omp for nowait
  for (i=0; i<n; i++) d[i] = e[i] + f[i];

  #pragma omp barrier

  scale = sum(a,0,n) + sum(d,0,n);
}
```

Collapse

- Clause pour les parallel for.
- Nombre de niveaux d'imbrication de boucle à aplatir afin d'augmenter le niveau de parallélisme possible.

OpenMP

SIMD

- Directive pour les boucles for indiquant que plusieurs itérations peuvent être exécutées concurremment avec des instructions SIMD (vectorielles).
- Il est possible d'avoir un combiné `#pragma omp for simd`.
- Toute fonction qui peut être appelée par du code SIMD doit être déclarée SIMD (`pragma omp declare SIMD`).
- Le compilateur devra s'assurer que le code visé pourra être compilé pour le processeur SIMD ciblé, par exemple avoir une copie en code natif et une copie en bytecode qui pourra être compilée à l'exécution pour le bon GPU.

Clauses

- safelen: la distance maximale entre deux itérations qui seront exécutées en parallèle sur le dispositif SIMD.
- simdlen: le nombre recommandé d'itérations à grouper en une instruction SIMD.
- aligned: à combien d'octets sont alignés les variables listées.

Distribute

- Directive pour les boucles afin de distribuer le travail entre les fils d'exécution primaires des équipes de la ligue courante.
- Stratégies de division du travail non précisée ou statique, `dist_schedule(static, chunk_size)`.
- Il est possible d'avoir un combiné
  `#pragma omp distribute simd` ou
  `#pragma omp teams distribute`.

Exemple

```
float sum = 0.0;
int i, i0;

#pragma omp target map(to: B[0:N], C[0:N]) map(tofrom: sum)
  #pragma omp teams thread_limit(block_threads) reduction(+:sum)
    #pragma omp distribute
    for (i0=0; i0<N; i0 += block_size)
      #pragma omp parallel for reduction(+:sum)
      for (i=i0; i< min(i0+block_size,N); i++)
        sum += B[i] * C[i];
```

Distribute parallel for

- Directive pour les boucles afin de distribuer le travail entre les fils d'exécution des équipes de la ligue courante.
- Stratégies de division du travail non précisée ou statique, `dist_schedule(static, chunk_size)`.
- Il est possible d'avoir un combiné `#pragma omp distribute parallel for simd`.

Loop

- Directive pour les boucles afin de distribuer le travail entre les fils d'exécution selon le contexte englobant (teams, parallel...).
- Les itérations peuvent être exécutées de manière concurrente dans n'importe quel ordre.
- On peut spéficier la portée de la division du travail (bind: teams / parallel / thread).
- Le compilateur choisit la meilleure stratégie (combinaison de distribute, parallel, simd...).
- Il est possible d'avoir un combiné `#pragma omp parallel loop`.

Exemple

```
float x[N], y[N];
float a = 2.0;

for(int i=0;i<N;i++){ x[i]=i; y[i]=0;}

#pragma omp parallel loop
for(int i = 0; i < N; ++i) y[i] = a*x[i] + y[i];
```

Exemple

```
// ecpannualmeeting.com 2020 Tutorial-with-ECP-template.pdf
#pragma omp target teams distribute // Now
for (i=0; i<N; ++i)
  #pragma omp parallel for
  for (j=0; j<N; ++j) x[j+N*i] *= 2.0;

#pragma omp target teams loop bind(teams) // Very soon
for (i=0; i<N; ++i)
  #pragma omp loop bind(thread)
  for (j=0; j<N; ++j) x[j+N*i] *= 2.0;

#pragma omp target // Soon
  #pragma omp loop bind(thread) collapse(2)
  for (i=0; i<N; ++i)
    for (j=0; j<N; ++j) x[j+N*i] *= 2.0;

#pragma omp loop bind(thread) collapse(2) // Later
for (i=0; i<N; ++i)
  for (j=0; j<N; ++j) x[j+N*i] *= 2.0;
```

Taskloop

```
#pragma omp taskloop [clause[[,] clause] ...]
loop-nest
```

- Les itérations de la boucle sont converties en tâches (task).
- Il est possible d'avoir un combiné `#pragma omp taskloop simd`.

Target

- Target: spécifier le dispositif cible à utiliser pour exécuter le bloc visé.
- Target data, target enter data, target exit data: spécifier les données à transférer vers / du dispositif avec la clause map.
- Target update: synchroniser certaines données entre l'hôte et le dispositif cible.
- Declare target, begin declare target, end declare target: définir certaines variables ou fonctions sur le dispositif cible.

Exemple

```
#define N 10000
#define M 1024
#pragma omp declare target
float Q[N][N];
#pragma omp declare simd uniform(i) linear(k) notinbranch
float P(const int i, const int k)
{ return Q[i][k] * Q[k][i]; }
#pragma omp end declare target

float accum(void) {
  float tmp = 0.0;
  int i, k;
  #pragma omp target
  #pragma omp parallel for reduction(+:tmp)
  for (i=0; i < N; i++) {
    float tmp1 = 0.0;
    #pragma omp simd reduction(+:tmp1)
    for (k=0; k < M; k++) tmp1 += P(i,k);
    tmp += tmp1;
  }
  return tmp;
}
```

Exemple

```
#define THRESHOLD1 1000000
#define THRESHOLD2 1000

extern void init(float*, float*, int);
extern void output(float*, int);

void vec_mult(float *p, float *v1, float *v2, int N) {
  int i;
  init(v1, v2, N);
  #pragma omp target if(N>THRESHOLD1) map(to: v1[0:N], v2[:N])\
      map(from: p[0:N])
  #pragma omp parallel for if(N>THRESHOLD2)
  for (i=0; i<N; i++) {
    p[i] = v1[i] * v2[i];
  }
  output(p, N);
}
```

Exemple

```
extern void init(float *, float *, int);
extern int maybe_init_again(float *, int);
extern void output(float *, int);
void vec_mult(float *p, float *v1, float *v2, int N) {
  int i;
  init(v1, v2, N);
  #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
  { int changed;
    #pragma omp target
    #pragma omp parallel for
    for (i=0; i<N; i++) p[i] = v1[i] * v2[i];
    changed = maybe_init_again(v1, N);
    #pragma omp target update if (changed) to(v1[:N])
    changed = maybe_init_again(v2, N);
    #pragma omp target update if (changed) to(v2[:N])
    #pragma omp target
    #pragma omp parallel for
    for (i=0; i<N; i++) p[i] = p[i] + (v1[i] * v2[i]);
  }
  output(p, N);
}
```

OpenMP

Fonctions de support: Threads

- omp_set_num_threads: spécifier le nombre de fils d'exécution.
- omp_get_num_threads: nombre de fils d'exécution.
- omp_get_max_threads: nombre maximal possible de fils d'exécution.
- omp_get_thread_num: numéro du fil courant.
- omp_get_num_procs: nombre de processeurs disponibles.
- omp_in_parallel: dans une région parallèle?

Fonctions de support: Threads

- omp_set_dynamic, omp_get_dynamic: ajustement dynamique du nombre de fils d'exécution.
- omp_set_nested, omp_get_nested: permission d'avoir une région parallèle imbriquée avec possiblement plus de fils d'exécution créés.
- OMP_STACKSIZE: taille des piles pour les fils d'exécution.
- omp_get_wtime: temps écoulé en secondes.
- omp_get_wtick: résolution de l'horloge utilisée pour omp_get_wtime.

Fonctions de support

- omp_set_schedule, omp_get_schedule: static, dynamic, guided, auto sont les stratégies possibles pour diviser les itérations de boucles entre les fils d'exécution, avec une granularité spécifiée.
- omp_get_thread_limit: nombre maximal de fils disponibles pour le programme.
- omp_set_max_active_levels, omp_get_max_active_levels: profondeur maximale d'imbrication de régions parallèles.
- omp_get_level, omp_get_active_level: profondeur courante de régions parallèles (actives) imbriquées.
- omp_get_ancestor_thread_num: numéro du thread parent.
- omp_get_team_size(level): nombre de thread dans la région parallèle ancêtre.

Fonctions de support: allocation mémoire

- On peut préciser comment allouer certaines variables avec la directive ou la clause `allocate`, et la clause `allocator`.
- On peut spécifier l'allocateur par défaut: omp_init_allocator, omp_destroy_allocator, omp_set_default_allocator, omp_get_default_allocator
- On peut spécifier un allocateur parmi: omp_default_mem_alloc, omp_large_cap_mem_alloc, omp_const_mem_alloc, omp_high_bw_mem_alloc, omp_low_lat_mem_alloc, omp_cgroup_mem_alloc, omp_pteam_mem_alloc, omp_thread_mem_alloc.

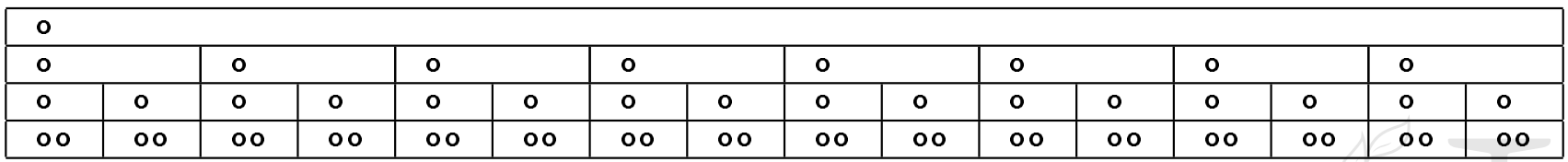
Placement des fils d'exécution

- Définir les *places* où peuvent être assignés les fils d'exécution.

```
OMP_PLACES = threads/cores/ll_caches/numa_domains/sockets
OMP_PLACES = place : length : stride, ...
OMP_PROC_BIND = true/false/primary/close/spread

setenv OMP_PLACES threads

setenv OMP_PLACES "{0:4}:4:4" # 4 places de 4 fils
setenv OMP_PLACES "{0:4},{4:4},{8:4},{12:4}" # idem
setenv OMP_PLACES \
  "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
```

Exemple

```
setenv OMP_PLACES "{0},{1},{2}, ... {29},{30},{31}"
# setenv OMP_PLACES threads (same as above if 32 threads)
sentenv OMP_NUM_THREADS "8,2,2"
sentenv OMP_PROC_BIND "spread,spread,close"

#pragma omp parallel // spread, 8 threads
  #pragma omp parallel // spread, 2 threads
    #pragma omp parallel // close, 2 threads
```



Exemple

```
# 4 sockets of 6 cores with 2 hyperthreads, round-robbin numbering among s/c/ht

# Thread per core
#pragma omp parallel num_threads(4*6) proc_bind(spread)

# Thread per socket
#pragma omp parallel num_threads(4) proc_bind(spread)
  # Thread per core
  #pragma omp parallel num_threads(6) proc_bind(spread)
    # Thread per hyperthread
    #pragma omp parallel num_threads(2) proc_bind(close)
```

Librairies de support

- OMPT: enregistrer des fonctions de rappel pour tous les événements importants reliés à OpenMP.
- Environ 40 événements comme parallel begin, parallel end, mutex acquire, mutex acquired, mutex released...
- Possibilité de générer une trace sur disque de ces événements OMPT.
- OMPD: interface pour des outils dans des processus séparés comme les débogueurs.
- Permet d'interroger le programme OpenMP sur les régions parallèles actives, les fils d'exécution, les tâches...

Outils de support

- Valgrind, Intel Thread Checker, Solaris Studio Thread Analyzer, Perf, Intel Vtune, LTTng.
- Vérification de la répartition de la charge entre les fils.
- Détection de blocages potentiels, vérifier l'ordre de prise des verrous.
- Détection de courses, valider que tous les accès écritures versus écritures ou lectures par des fils différents sont séparés par des synchronisations qui assurent un ordonnancement prévisible.
- Vérification de faux partage de lignes de cache.

OpenMP

Précautions

- Attention au placement des variables partagées pour éviter le faux partage en cache.
- S'assurer que les fonctions appelées dans les sections parallèles sont prévue pour cela (thread safe).
- Accéder en C/C++ les matrices par colonne (dernier indice qui varie le plus vite).
- Choisir des morceaux assez gros pour minimiser le surcoût de contrôle (et aider la cache) mais assez petit pour permettre d'équilibrer le travail de chaque fil.