Automatic Collapsing of Non-Rectangular Loops

Philippe Clauss, Ervin Altıntas, Matthieu Kuhn

Automatic Collapsing of Non-Rectangular Loops

Philippe Clauss and Ervin Altıntaş
Inria CAMUS, ICube Lab., CNRS
University of Strasbourg
Illkirch, France
{*altintas,clauss*}*@unistra.fr*

Matthieu Kuhn
Inria HIEPACS
Inria Bordeaux - Sud-Ouest
Talence, France
matthieu.kuhn@inria.fr

***Abstract*—Loop collapsing is a well-known loop transformation which combines some loops that are perfectly nested into one single loop. It allows to take advantage of the whole amount of parallelism exhibited by the collapsed loops, and provides a perfect load balancing of iterations among the parallel threads.**

However, in the current implementations of this loop optimization, as the ones of the OpenMP language, automatic loop collapsing is limited to loops with constant loop bounds that define rectangular iteration spaces, although load imbalance is a particularly crucial issue with non-rectangular loops. The OpenMP language addresses load balance mostly through dynamic runtime scheduling of the parallel threads. Nevertheless, this runtime schedule introduces some unavoidable execution-time overhead, while preventing to exploit the entire parallelism of all the parallel loops.

In this paper, we propose a technique to automatically collapse any perfectly nested loops defining non-rectangular iteration spaces, whose bounds are linear functions of the loop iterators. Such spaces may be triangular, tetrahedral, trapezoidal, rhomboidal or parallelepiped. Our solution is based on original mathematical results addressing the inversion of a multi-variate polynomial that defines a ranking of the integer points contained in a convex polyhedron.

We show on a set of non-rectangular loop nests that our technique allows to generate parallel OpenMP codes that outperform the original parallel loop nests, parallelized either by using options "static" or "dynamic" of the OpenMP-schedule clause.

***Keywords*-loop parallelization; loop collapsing; load balancing; OpenMP.**

I. Introduction

Loop collapsing – alternatively called *loop coalescing* or *loop flattening* [1], [2] – combines two or more loops into a single loop, producing less loop overhead, better load balance and exposing more concurrency when the collapsed loops are parallel, and can improve the opportunities for other optimizations, such as loop unrolling and vectorization.

Since version 3.0 of OpenMP [3], the `collapse` clause may be used in the directive `omp for` to specify how many loops are associated with the current loop construct. The parameter of the `collapse` clause must be a constant positive integer expression. If no `collapse` clause is present, the only loop that is associated with the loop construct is the one that immediately follows the loop directive. If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space. The iteration count for each associated loop is computed before entry to the outermost loop. If execution of any associated loop changes any of the values used to compute any of the iteration counts, then the behavior is unspecified. Hence the OpenMP `collapse` clause only handles perfectly nested rectangular loops. In the newsletter of the OpenMP Architecture Review Board, January 2003, one can read: *We have not concluded whether collapsing of non-rectangular loops, and/or non-perfectly nested loops can be sufficiently easily handled/specified. This is still the subject of discussion*; and in the newsletter of February 2003: *The committee has looked at automatic collapsing of non-rectangular loops, and has decided NOT to recommend this for addition in OpenMP 3.0.*

However, performance issues due to load imbalance are typical for non-rectangular iteration spaces. The reason why non-rectangular loops are not handled is related to the recovery of the original loop indices. Indeed, when collapsing loops, original indices are reduced to a single index. Thus, every reference to the original indices made by the loop statements requires to recover their values from the current value of the single index resulting from collapsing the original loops. When the original loop bounds are constant, such a recovery is straightforward: without loss of generality, consider l loops whose respective index i_k ranges from 0 to $N_k - 1$, $k = 1..l$. Collapsing these l loops results into one single loop whose index i ranges from 1 to $N_1 \times N_2 \times ... \times N_l$. Each original index i_k can be recovered from i in the following way:

$$
\begin{aligned}
i_1 &= \lfloor \frac{i}{N_2 \times N_3 \times ... \times N_l} \rfloor \\
i_2 &= \lfloor \frac{i \ mod \ (N_2 \times N_3 \times ... \times N_l)}{N_3 \times N_4 \times ... \times N_l} \rfloor \\
i_3 &= \lfloor \frac{(i \ mod \ (N_2 \times N_3 \times ... \times N_l)) \ mod \ (N_3 \times N_4 \times ... \times N_l)}{N_4 \times N_5 \times ... \times N_l} \rfloor \\
&\dots
\end{aligned}
$$

where $\lfloor x \rfloor$ denotes the integer part (floor) of x. Thus, an implementation of loop collapsing mostly consists in embedding these recovery computations in the generated

code.

For non-rectangular loops, two main issues are not handled in the current implementations:

- total number of iterations: when collapsing loops, the bound of the resulting single loop is the total number of iterations of the original loop nest. When the original bounds are not constant, but depending linearly on some surrounding loop indices and unknown parameters, this bound requires the computation of the exact number of integer points contained in the original iteration domain.
- original indices recovery: recovering the values of the original indices, solely from the unique loop index of the collapsed loops, requires to invert a particular multivariate polynomial. This polynomial, called *ranking polynomial*, associates to each tuple of original indices, an integer which is the rank of the associated iteration among all iterations.

In this paper, we solve both issues to enable the application of loop collapsing to non-rectangular loops, characterized by bounds which are linear functions of the surrounding loop iterators. More generally, we handle the mathematical problem of lexicographic ranking and unranking functions [4] for integer points contained in parametrized polyhedra, and apply it to parallel loop scheduling.

The paper is organized as follows. In the next Section, an example of a correlation computation defining a triangular loop nest, which is collapsed using our technique, is used to motivate the paper. In Section III, we recall the notion of *ranking Ehrhart polynomial* which is central to the proposed method. Section IV explains the mathematical aspects of the proposed technique, which are related to the inversion of ranking Ehrhart polynomials, and also discusses its limitations. The time-overhead related to index recovery is addressed in Section V, while Section VI addresses specific issues related to vectorization and execution on GPU processors. Experiments, presented in Section VII, highlight the significant time improvements provided by collapsing non-rectangular loops parallelized with OpenMP. Section VIII discusses some related work, while conclusions are given in Section IX.

II. Motivating example

Consider the loop nests in Figure 1, which is part of a correlation computation. Loops i and j do not carry any dependence and can then be parallelized. However, since the lower bound of the j-loop is not constant, but depending on the current value of index i, both i and j loops cannot be collapsed with OpenMP. Hence, one user may adopt another parallelization strategy as nested parallelism or dynamic loop scheduling. However each of them involves specific time-overhead issues.

A first solution is to parallelize the outermost loop using the OpenMP directive `#pragma omp for schedule(static)`: equal slices of the i-loop are then distributed to the parallel threads. However, since the number of iterations of the j-loop is proportional to the value of index i, the first i-loop slices, covering the lowest values of i, contain far more iterations than the last slices. This situation yields an important and penalizing load imbalance among the threads, as illustrated by Figure 2, where the total execution time of the parallel loop is determined by the time of thread 0.

```
#pragma omp parallel for private(j,k) schedule(static)
for (i=0 ; i < N-1 ; i++)
 for(j=i+1 ; j<N ; j++) {
  for(k=0 ; k<N ; k++)
   a[i][j]+=b[k][i]*c[k][j];
  a[j][i] = a[i][j]; }
```

Figure 1. Correlation computation

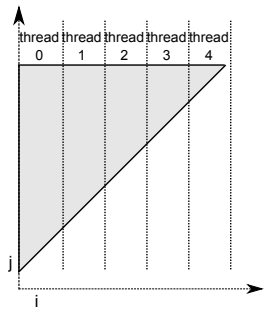

Figure 2. Unbalanced distribution of iterations among 5 threads of the correlation iteration domain using static OpenMP schedule

A possibly better solution may be the OpenMP schedule clause `dynamic`. However, even if better performance may be observed with this parallelization scheme, dynamic distribution of the iterations among the threads involves necessarily some runtime overhead, particularly when a huge number of parallel threads have to be supplied with iterations that are numerous. Such a solution is generally not scalable.

Another solution would be to take advantage of the nested parallelism of the i and j loops, by enabling the dynamic creation of additional parallel threads for every value of i, at each execution of the j-loop. In that case as well, the runtime overhead of dynamic scheduling may be too penalizing, because of the recurring creations of new threads and the resulting overloading of the operating system.

Consider now the loop nest of Figure 3. Both i and j loops have been collapsed into one single loop running $\frac{(N-1)N}{2}$ iterations. The original indices i and j are recovered at each iteration from the single loop index pc, by computing the

```
#pragma omp parallel for private(i,j,k) schedule(static)
for (pc=1 ; pc <= (N-1)*N/2 ; pc++) {
 i=floor(-(sqrt(4*N*N-4*N-8*(double)pc+9)-2*N+1)/2);
 j=floor(-(2*(double)i*N-2*(double)pc-(double)i*(double)i
   -3*(double)i)/2);
 for(k=0 ; k<N ; k++)
  a[i][j]+=b[k][i]*c[k][j];
 a[j][i]=a[i][j]; }
```

Figure 3. Collapsed correlation computation

```
first_iteration = 1;
#pragma omp parallel for private(i,j,k) \
 firstprivate(first_iteration) schedule(static)
for (pc=1 ; pc <= (N-1)*N/2 ; pc++) {
 if (first_iteration)
 {
   i = floor(-(sqrt(4*N*N-4*N-8*(double)pc+9)-2*N+1)/2);
   j = floor(-(2*(double)i*N-2*(double)pc
       -(double)i*(double)i-3*(double)i)/2);
   first_iteration = 0;
 }
 for(k=0 ; k<N ; k++)
  a[i][j]+=b[k][i]*c[k][j];
 a[j][i] = a[i][j];
 j++;
 if (j>=N) {i++; j=i+1;}
}
```

Figure 4. Collapsed correlation computation with reduced control time-overhead

following arithmetic expressions at each iteration:

$$i = \left\lfloor -\frac{\sqrt{4\,N^2 - 4\,N - 8\,pc + 9} - 2\,N + 1}{2} \right\rfloor$$

$$j = \left\lfloor -\frac{2\,i\,N - 2\,pc - i^2 - 3\,i}{2} \right\rfloor$$

This single pc-loop can be parallelized with OpenMP using its schedule clause `static`, to take advantage of the nested parallelism of the original i and j loops, and to get a well balanced load of iterations (i, j) among the parallel threads.

Notice that a naive alternative solution would be to reproduce the original computations of the loop indices inside the pc-loop, by incrementing them relatively to the original loop bounds. But this strategy is not suitable for parallelization, since the starting values of the original indices for each parallel thread cannot be determined, while only the local value of the new iterator pc is known by each thread.

In our solution, the recovery of the original i and j indices requires the computation of a square root, a floor and some polynomials depending on pc, at each iteration. Such a computation at every iteration may introduce a penalizing time-overhead. We reduce it significantly by performing the costly computation only once per threads, at the first iteration, and by computing the next indices values by reproducing the standard indices incrementation of the original loop nest. The resulting program is shown in Figure 4.

Both next Sections address the required mathematical background and the proposed technique for collapsing automatically non-rectangular loops.

III. Ranking Ehrhart polynomials

Ehrhart polynomials were originally proposed and extended to program analysis in [5]. These integer-valued polynomials express the exact number of integer points contained in a finite multi-dimensional convex polyhedron which depends linearly on integer parameters. They have many applications for the quantitative analysis of loop nests whose loop bounds are linear functions of the surrounding loop indices and integer parameters, and which statements are referencing multi-dimensional array elements through linear functions of the loop indices and parameters. Such a counting of integer points may translate to the exact number of iterations of a parameterized loop nest, the exact number of memory locations touched by a loop nest, the maximum number of parallel iterations, etc. When considering a d-dimensional polyhedron – as for example the iteration space of a d-depth loop nest – depending linearly on integer parameters $p_1, p_2, \ldots, p_m$, its Ehrhart polynomial is a polynomial of degree d whose variables are $p_1, p_2, \ldots, p_m$.

Ehrhart polynomials can be automatically computed using existing algorithm implementations as the one of the polyhedral library PolyLib [6] or the one of the barvinok library [7].

Among their applications, Ehrhart polynomials are used in [8] to reorganize the memory layout of array elements accessed by a loop nest, in order to improve their spatial data locality: array elements are relocated in memory in the same order as they are accessed. In this approach, the new location of an array element is given by the order, or *rank*, of the iteration referencing it.

Such a rank of iterations is given by a polynomial, called a *ranking polynomial*, whose variables are the loop indices, and whose evaluation results in the number of iterations preceding a given iteration. More formally, the *ranking polynomial* of a d-depth loop nest whose loop indices, from the outermost to the innermost, are $(i_1, i_2, \ldots, i_d)$, is denoted $r(i_1, i_2, \ldots, i_d)$. Without loss of generality, if $(0, 0, \ldots, 0)$ defines the first iteration of the loop nest, then $r(0, 0, \ldots, 0) = 1$, $r(0, 0, \ldots, 1) = 2$, and so on. If $(N_1, N_2, \ldots, N_d)$ are the indices of the very last iteration, then $r(N_1, N_2, \ldots, N_d)$ is the total number of iterations of the loop nest.

The computation of the ranking polynomial of a loop nest is detailed in [8]. We recall this technique by applying it to the correlation computation of Figure 1. Let us compute the ranking polynomial of the outer i and j loops, $r(i, j)$. The rank of a given iteration (i_0, j_0) is equal to the number of iterations that are executed before (i_0, j_0) (included), i.e., the number of couples (i, j) inside the iteration domain which are lexicographically less than or equal to (i_0, j_0):

$$\forall (i_0, j_0) \text{ s.t. } 0 \leq i_0 < N-1 \text{ and } i_0+1 \leq j_0 < N,$$

$$r(i_0, j_0) = \#\{(i,j) \mid (i,j) \trianglelefteq (i_0, j_0), \\ 0 \leq i < N-1, \\ i+1 \leq j < N\}$$

where $\trianglelefteq$ denotes the lexicographic order. Since lexicographic inequalities are not linear, the problem is split as the conjunction of two equivalent sets of linear inequalities, according to the definition of the lexicographic order:

$$(i,j) \trianglelefteq (i_0, j_0) \Leftrightarrow (i < i_0) \text{ or } (i = i_0 \text{ and } j \leq j_0)$$

Therefore, the sets whose integer points must be counted can be defined as the union of two disjoint convex polyhedra, and $r(i_0, j_0)$ as the sum of two Ehrhart polynomials:

$$\begin{aligned} r(i_0, j_0) &= \#\{(i,j) \mid 0 \leq i < i_0, i+1 \leq j < N\} \\ &\quad + \#\{(i,j) \mid i = i_0, i_0+1 \leq j \leq j_0\} \\ &= \frac{i_0 \ (2\,N - i_0 - 1)}{2} + (j_0 - i_0) \\ &= \frac{2\,i_0\,N + 2\,j_0 - {i_0}^2 - 3\,i_0}{2} \end{aligned}$$

One can verify that the rank of the first iteration $(0,1)$, $r(0,1)$, is equal to 1, the rank of the second iteration $r(0,2) = 2$, the rank of the third iteration $r(0,3) = 3$ and so on. The rank of the last j-iteration when $i = 0$, $r(0, N-1) = N-1$, and the rank of the first iteration when $i = 1$, $r(1,2) = N$. The total number of iterations is $r(N-2, N-1) = \frac{(N-1)\,N}{2}$.

Notice the following important property: such a ranking polynomial associates, to each iteration index tuple, a unique integer of a continuous interval of integers starting at 1. This continuous interval is the range of integers between one and the total number of iterations. Conversely, each integer value in the interval is associated to one unique iteration. Thus, a ranking polynomial defines a *bijection* between the iteration domain and the interval of successive integers. It can also be seen as the one-dimensional polynomial schedule function of the iterations, which is equivalent to the original multi-dimensional linear schedule defined by the nested loops.

Another important property is that such a ranking polynomial is monotonically increasing over the integers, from 1 to the total number of iterations, relatively to the lexicographic order of the loop indices.

IV. Collapsing non-rectangular loops

The collapsing technique presented in this paper is devoted to loop nests whose loops to be collapsed: (1) are perfectly nested, (2) do not carry any dependence, and (3) have one unique iterator, *whose loop bounds are linear combinations of the surrounding loops' iterators*; the linear combinations may also depend linearly on unknown parameters, which are typically size parameters. Such loop bounds may define non-rectangular iteration spaces that are triangular, tetrahedral, trapezoidal, rhomboidal or parallelepiped. The collapsing of loops carrying dependences and the collapsing of imperfect loop nests are beyond the scope of this paper. Thus, our model of loops that may be collapsed can be depicted in Figure 5, where $l_k(i_1, i_2, ..., i_{k-1})$ and $u_k(i_1, i_2, ..., i_{k-1})$ denote linear combinations of iterators $i_1, i_2, ..., i_{k-1}$, and $\mathbf{S}(i_1, i_2, ..., i_c)$ denotes a sequence of statements whose instances depend on iterators $i_1, i_2, ..., i_c$.

for $(i_1 = l_1 \ ; \ i_1 < u_1 \ ; \ i_1 ++)$
  for $(i_2 = l_2(i_1) \ ; \ i_2 < u_2(i_1) \ ; \ i_2 ++)$
    for $(i_3 = l_3(i_1, i_2) \ ; \ i_3 < u_3(i_1, i_2) \ ; \ i_3 ++)$
      ...
        for $(i_c = l_c(i_1, i_2, ..., i_{c-1}) \ ;$
            $i_c < u_c(i_1, i_2, ..., i_{c-1}) \ ; \ i_c ++)$
          $\{ \ \mathbf{S}(i_1, i_2, ..., i_c); \ \}$

Figure 5. Model of non-rectangular loops that are handled

A. Unranking functions as inverse ranking polynomials

Consider a loop nest of d perfectly nested loops that may be non-rectangular. Suppose that we would like to collapse c successive loops of this nest ($c \leq d$), whose loop indices are $i_1, i_2, \ldots, i_c$, from the outermost to the innermost.

The ranking Ehrhart polynomial of the sub-nest made of these c loops, $r(i_1, \ldots, i_c)$, is a multi-variate polynomial of degree c. Without loss of generality, assume that every loop's lower bound is equal to 0, and that the tuple of indices $(N_1, \ldots, N_c)$ corresponds to the very last iteration of the c-depth sub-nest. Then, $r(0, 0, \ldots, 0) = 1$, since $(0, 0, \ldots, 0)$ corresponds to the first iteration, and the total number of iterations of the single loop resulting from collapsing the c loops is equal to $r(N_1, \ldots, N_c)$. Let pc be the loop index of the single loop resulting from collapsing, its header can be generated as being:

```
for (pc = 1 ; pc <= r(N1,...,Nc) ; pc++)
```

Notice that the successive values taken by index pc are the successive values of the ranking polynomial $r(i_1, ..., i_c)$. The main issue is now to recover the original indices $(i_1, \ldots, i_c)$ from the single index pc.

Mathematically speaking, the problem is to reverse the ranking polynomial $r(i_1, \ldots, i_c)$ in order to find, for any given value pc_0 reached by iterator pc, the tuple $(i_1, \ldots, i_c)$ such that $r(i_1, \ldots, i_c) = pc_0$. As highlighted in the previous Section, any ranking polynomial defines a bijection from the iteration domain to the range of integers between one and the total number of iterations, and thus it is theoretically invertible. It is also monotonically increasing relatively to the lexicographic order of the tuples $(i_1, \ldots, i_c)$.

Consider first the outermost loop and the symbolic univariate polynomial equation:

$$r(x_1(pc_0), 0, \ldots, 0) - pc_0 = 0$$

where $x_1(pc_0)$ denotes its symbolic solution which is parametrized by pc_0. Depending on the degree c of the ranking polynomial, the polynomial $r(x_1(pc_0), 0, \ldots, 0) - pc_0$ may have c real or complex solutions.

Among these solutions, only one solution is such that $\lfloor x_1(1) \rfloor = 0$, i.e., such that the computed index $i_1 = \lfloor x_1(1) \rfloor$ for the very first iteration is, as expected, equal to 0. Moreover, we are certain that such a solution exists, by definition of the ranking polynomial. Nevertheless, it is shown in the next Subsection that in some cases, the symbolic roots of the polynomial may require the computation of a complex number whose imaginary part is null.

Since pc_0 may be the rank of any iteration of the collapsed loop nest, value $r(i_1 = \lfloor x_1(pc_0) \rfloor, 0, \ldots, 0)$ may not be exactly equal to pc_0, but such that:

$$r(i_1, 0, \ldots, 0) \leq pc_0 < r(i_1 + 1, 0, \ldots, 0)$$

Since the ranking polynomial is monotonically increasing, i_1 is the value of the outermost index such that there exists a unique tuple $(i_2, ..., i_c)$ in the iteration domain such that:

$$r(i_1, i_2, \ldots, i_c) = pc_0$$

We now propagate this first solution i_1 in order to find the next index value i_2. For this purpose, we solve the equation:

$$r(i_1, x_2(i_1, pc_0), 0, \ldots, 0) - pc_0 = 0$$

where $x_2(i_1, pc_0)$ denotes its solution, which is also parametrized by the surrounding loop index i_1, since in non-rectangular loop nests, the value of index i_2 may depend on i_1. As before, we select the solution $x_2(i_1, pc_0)$ which is such that $\lfloor x_2(0, 1) \rfloor = 0$ and set $i_2 = \lfloor x_2(i_1, pc_0) \rfloor$.

The same propagation and solving process is repeated, where each solution depends on all its surrounding loop indices and index pc_0, until the computation of the last index i_c, which is simply deduced as being:

$$i_c = pc_0 - r(i_1, i_2, \ldots, i_{c-1}, 0)$$

Finally, the single loop resulting from collapsing the c nested loops can be generated as follows:

for (pc = 1 ; pc <= $r(N_1, \ldots, N_c)$; pc++) {

```
/* recovery of the original indices */
```
$i_1 = \lfloor x_1(pc) \rfloor$;
$i_2 = \lfloor x_2(i_1, pc) \rfloor$;
...
$i_{c-1} = \lfloor x_{c-1}(i_1, i_2, \ldots, i_{c-2}, pc) \rfloor$;
$i_c = pc - r(i_1, i_2, \ldots, i_{c-1}, 0)$;

```
/* original statements */
statements(i1, ..., ic); }
```

More generally, when lower bounds are non-null integers, the recovery of index i_k requires the convenient symbolic root of:

$$r(i_1, ..., x_k(i_1, ..., i_{k-1}, pc_0), lb_{k+1}, lb_{k+2}, ..., lb_c) - pc_0 = 0$$

where lb_q denotes the lower bound of index i_q. Such a lower bound can be a integer constant, or a linear combination of the surrounding loop indices. As an example, notice that in the correlation computation of Figure 1, the lower bound of index j is $i + 1$. In such cases, the lower bounds $lb_{k+1}, lb_{k+2}, ..., lb_c$ must be set to their lexicographic minimums parametrized by $i_1, ..., i_k$. Parametric lexicographic minimums can be computed using library ISL [9].

To illustrate this technique, we apply it now to the correlation computation of Figure 1, and show how the flattened loops of Figure 3 and 4 have been generated. As presented in Section III, the ranking polynomial of loops i and j is:

$$r(i, j) = \frac{2\,i\,N + 2\,j - i^2 - 3\,i}{2}$$

from which the upper bound of the single loop resulting from collapsing is deduced: $r(N-2, N-1) = \frac{(N-1)\,N}{2}$. We first solve equation $r(i(pc), i(pc)+1) - pc = 0$. Notice that the second index is set to the lexicographic minimum value of j parametrized by $i(pc)$.

Any computer algebra system may be used to compute the symbolic roots, as for example Maxima[1]:

```
(%i1) r(i,j):=(2*i*N+2*j-i^2-3*i)/2$
(%i2) solve(r(i,i+1)-pc,i);
                              2
                  sqrt(4 N  - 4 N - 8 pc + 9) - 2 N + 1
(%o2) [i = - ---------------------------------------,
                                2
                           2
                  sqrt(4 N  - 4 N - 8 pc + 9) + 2 N - 1
         i =  ---------------------------------------]
                                2
```

To select the convenient root which is such that $\lfloor i(1) \rfloor = 0$, we evaluate them both for $pc = 1$:

```
(%i3) i1(pc):=-(sqrt(factor(4*N^2-4*N-8*pc+9))-2*N+1)/2$
(%i4) i2(pc):=(sqrt(factor(4*N^2-4*N-8*pc+9))+2*N-1)/2$
(%i5) expand(i1(1));
(%o5) 0
(%i6) expand(i2(1));
(%o6) 2*N-1
```

The computation of index j is straightforward. The formula can be obtained in the following way with Maxima:

```
(%i7) solve(r(i,j)-pc,j);
                                  2
                  2 i N - 2 pc - i  - 3 i
(%o7) [j = - ------------------------]
                          2
```

Finally, we get:

$$i = \left\lfloor -\frac{\sqrt{4N^2 - 4N - 8pc + 9} - 2N + 1}{2} \right\rfloor$$

$$j = \left\lfloor -\frac{2iN - 2pc - i^2 - 3i}{2} \right\rfloor$$

[1]http://maxima.sourceforge.net

B. Limitations of the method

Only polynomial equations whose degree is at most equal to 4 can be solved symbolically, with exact expressions for roots. The handled uni-variate polynomial equations are built from a multi-variate ranking polynomial, where one index i_k is set as the equation unknown, indices $i_1, ..., i_{k-1}$ are set as symbolic parameters, and indices $i_{k+1}, ..., i_c$ are set to their lexicographic minimum values parametrized by $i_1, ..., i_k$. Thus, to ensure that such a built equation has a degree less than 4, the ranking polynomial must be such that any index i_k, in any of its monomials, has a degree less than 4, *i.e.*, any monomial is of the form: $a\, i_1^{p_1} i_2^{p_2} ... i_c^{p_c}$ where a is a rational number, and every power p_k is such that $0 \leq p_k \leq 4$.

Loop nests yielding such ranking polynomials are such that the maximum number of loops, whose loop trip counts depend on a given index i_k, is less than or equal to 4. For example, both outermost loops of the correlation computation in Figure 1 depend on index i, yielding a ranking polynomial where index i is of power 2 in some monomial. In the example of Figure 6, all the three loops depend on index i, and the two innermost loops depend on index j, yielding a ranking polynomial where index i is of power 3 in some monomial, and index j is of power 2.

However, notice that the dependence of loop trip counts regarding indices is transitive: if a loop index j depends on a surrounding loop index i, and a inner loop index k depends on j, then k depends also on i: index i is of power 3, in some monomial of the ranking polynomial. Notice also that loops may depend simultaneously on several surrounding loops' indices, as for example in `for(k=0;k<i+j;k++)`.

Loops with constant bounds, whose indices are not used in any other loop's bounds, always yield monomials where their indices are of power one. Thus, the non-rectangular loop nests that can be handled by using our method may be of any depth, but are such that the number of nested loops that all depend on a given index is less than or equal to 4. This should be quite sufficient for most cases, regarding the usual loop nest depth and complexity of user codes.

One may believe that our technique could be applied recursively, by collapsing non-rectangular loops by successive pairs of loops, either from the innermost to the outermost loop, or in the opposite way, in order to handle only polynomial equations of degree 2. Unfortunately, both ways yield non-linear issues:

- from innermost to outermost loop: collapsing a pair of innermost non-rectangular loops results in a single loop whose upper bound is a non-linear polynomial depending on the surrounding loop indices. Hence, the ranking polynomial of the next surrounding loop and the new single loop cannot be computed.
- from outermost to innermost loop: after having collapsed the first pair of outermost non-rectangular loops, the ranking polynomial of the resulting single loop and the next inner loop cannot be computed, since the bounds of this latter loop depend on indices that have been collapsed, and whose values are determined by non-linear expressions, which are symbolic roots of the previous ranking polynomial.

C. Complex or real roots

Intuitively, one may believe that among the computed roots, the selection of the convenient root should be achieved solely among the real roots. However, since the computed roots are symbolic, they may alternatively be complex or real, depending on the value of pc.

As an illustrative example, consider the loop nest of depth 3 in Figure 6, where `S(i,j,k)` denotes a sequence of statements depending on indices i, j and k. We first compute the ranking polynomial of the nest:

$$r(i,j,k) = \frac{6\,k - 3\,j^2 + 6\,i\,j + 3\,j + i^3 + 3\,i^2 + 2\,i + 6}{6}$$

Hence the total number of iterations is:

$$r(N-2, N-2, N-2) = \frac{N^3 - N}{6}$$

First, we solve equation $r(i(pc), 0, 0) - pc = 0$ and get, among the three roots, a seemingly-real root which is:

$$\left(\frac{\sqrt{243\,pc^2 - 486\,pc + 242}}{3^{\frac{3}{2}}} + 3\,pc - 3\right)^{\frac{1}{3}} + \frac{1}{3\left(\frac{\sqrt{243\,pc^2-486\,pc+242}}{3^{\frac{3}{2}}} + 3\,pc - 3\right)^{\frac{1}{3}}} - 1$$

Instantiating pc with 1 yields a complex root since $\sqrt{243 \times 1^2 - 486 \times 1 + 242} = \sqrt{-1} = I$. However, the whole computation of the root for $pc = 1$ results in the complex number $0 + 0 \times I = 0$, which is the right solution. Moreover, the root becomes real for any value of pc strictly above 1.

This has two main consequences:

- The selection of the convenient root must not be done relatively to its type (complex or real), but relatively to the correctness of the values it provides;
- In the generated code, the indices should be computed by using complex variables and mathematical functions, since float functions may return `NaN` (Not a Number).

By solving $r(i, j(pc), 0) - pc = 0$, we get the following solution for $j(pc)$:

$$-\frac{\sqrt{3}\,\sqrt{-24\,pc + 4\,i^3 + 24\,i^2 + 44\,i + 51} - 6\,i - 9}{6}$$

Finally, by solving $r(i, j, k(pc)) - pc = 0$, we get for k:

$$\frac{6\,pc + 3\,j^2 - (6\,i + 3)\,j - i^3 - 3\,i^2 - 2\,i - 6}{6}$$

```
for (i = 0 ; i < N-1 ; i++)
 for(j = 0 ; j < i+1 ; j++)
  for(k = j ; k < i+1 ; k++)
   S(i,j,k);
```

Figure 6. 3-depth loop nest example

```
for (pc=1 ; pc <= pow(N,3)/6-N/6 ; pc++) {
 i=floor(creal(cpow(pow(3.0,-3.0/2)*csqrt(243
   *pow((double)pc,2.0)-486*(double)pc+242)
   +3*(double)pc-3,1.0/3)+cpow(pow(3.0,-3.0/2)
   *csqrt(243*pow((double)pc,2.0)-486*(double)pc+242)
   +3*(double)pc-3,-1.0/3)/3-1));
 j=floor(-(sqrt(3.0)*sqrt((-24*(double)pc)
   +4*pow((double)i,3.0)+24*pow((double)i,2.0)
   +44*(double)i+51)-6*(double)i-9)/6);
 k=floor((6*(double)pc+3*pow((double)j,2.0)+((-3)
   -6*(double)i)*j-pow((double)i,3.0)-3*pow((double)i,2.0)
   -2*(double)i-6)/6);
 S(i,j,k); }
```

Figure 7. Collapsed 3-depth loop nest

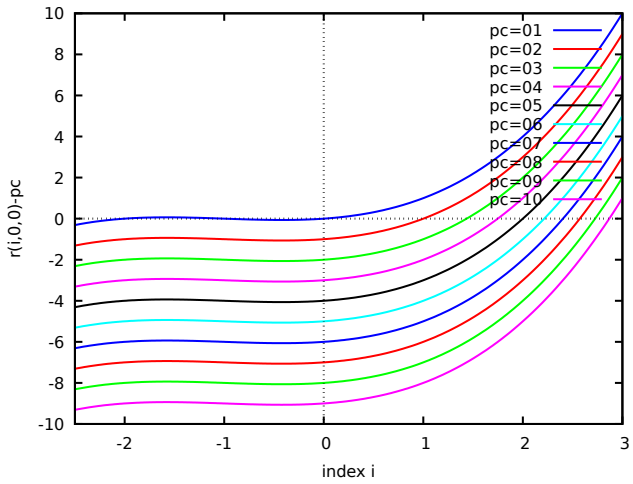

Figure 8. $r(i,0,0) - pc$ for $i = -2.5..3$ and $pc = 1..10$

The resulting flattened loop is showed in Figure 7, where the complex mathematical functions `creal`, `cpow` and `csqrt` are invoked to recover the original indices.

D. Uniqueness of the convenient symbolic root

A remaining question is whether the convenient symbolic root is unique. Ranking polynomials $r(i_1,...,i_d)$ are monotonically increasing relatively to the lexicographic order of the loop indices $(i_1,...,i_d)$. So are the intermediate polynomials $r(i_1,0,...,0), r(i_1,i_2,...,0),...,r(i_1,...,i_{d-1},0)$ that are used to get the symbolic roots for each index. Subtracting pc to each of these polynomials to set the equations has the geometrical effect of translating their respective curves along the pc-axis, such that it crosses the index-axis at its roots. Thus, all the curves, for all values of pc, are parallel, and the number, order and types of the symbolic roots remain unchanged, whatever the value of pc. On Figure 8, curves of the polynomials $r(i,0,0) - pc$ of the previous example of Figure 6 are represented for illustration.

V. Minimization of the index rediscovery cost

After having collapsed non-rectangular loops, the computation of the original indices values from the unique index pc may impose a penalizing time-overhead due to complex or floating-point operations and invocations to `floor`, `sqrt` and `pow`. We solve this issue by computing indices using these costly operations only once per threads, or more generally, once per chunks of iterations that are distributed among the threads. For each chunk, the floating-point operations to recover indices are only used at the very first iteration, while they are recovered for the next iterations in the same way as in the original sequential non-collapsed loop nest through successive incrementations.

An OpenMP implementation using `static` scheduling is the following:

```
first_iteration = 1;
#pragma omp parallel for firstprivate(first_iteration) \
        private(Indices) schedule(static)
for (pc = 1 ; pc <= N ; pc++) {
 if (first_iteration) {
  Costly_Recovery(pc);
  first_iteration = 0; }
 Statements(Indices);
 Incrementation(Indices); }
```

This scheme has been applied on the loop nest of Figure 4. When distributing chunks of iterations of size `CHUNK`, an OpenMP implementation is:

```
#pragma omp parallel for private(Indices) \
        schedule(static, CHUNK)
for (pc = 1 ; pc <= N ; pc++) {
 if ((pc-1) % CHUNK==0) {
  Costly_Recovery(pc); }
 Statements(Indices);
 Incrementation(Indices); }
```

Obviously, `dynamic` scheduling requires indices to be recovered by evaluating the roots at each iteration, with maximum computation cost, although the usage of such a schedule with collapsed loops does not really make sense.

VI. On vectorization and GPU execution

A. Vectorization

Let *vlength* be the maximum number of elements that may be computed simultaneously, according to the size of the vector registers. Vectorization requires to exhibit *vlength* independent iterations of the target loop nest. When non-rectangular parallel nested loops have been collapsed through our technique, *vlength* consecutive iterations of the resulting single loop may be vectorized. However, the values of the original indices, that have to be recovered for *vlength* consecutive iterations, are generally not only related through successive incrementations of the original innermost index. A costly recovery performed at each of the *vlength* vectorized iterations would obviously be prohibitive.

In order to perform the costly recovery only once per thread, a solution is to pre-compute *vlength* tuples of original indices by *vlength* successive incrementations. A general scheme could be the following:

```
#pragma omp parallel for firsprivate(first_iteration) \
            private(i,j,k,T) schedule(static)
for (pc = 1 ; pc <= N ; pc += vlength) {
  if (first_iteration) {
    CostlyRecovery(pc);
    first_iteration = 0; }
  for (v = pc ; v <= min(pc+vlength-1, N) ; v++) {
    T[v-pc] = Indices;
    Incrementation(Indices); }
/* Vectorization */
#pragma omp simd
  for (v = pc ; v <= min(pc+vlength-1, N) ; v++)
    Statements(T[v-pc]);
}
```

where array `T` is a thread private array of size *vlength*.

B. GPU execution

When programming GPU processors, a good strategy is to distribute consecutive iterations among the threads of a same warp, in order to achieve memory coalescing. Let W be the number of threads per warp. With such a distribution, each thread will run iterations that are spaced by W original consecutive iterations. When parallel nested loops have been previously collapsed with our technique, each thread can recover the next values of the original indices by incrementing W times the current indices values, while performing the costly recovery only once. A general scheme could be the following:

```
/* parallel threads in a warp */
for (thread = 0 ; thread < W ; thread++){
  for (pc = thread+1 ; pc <= N ; pc += W){
    if (pc == thread+1) CostlyRecovery(pc);
    Statements(Indices);
    for(inc = 0 ; inc < W ; inc++)
      Incrementation(Indices); }}
```

VII. Experiments

We have developed a software tool taking as input C source codes where non-rectangular loop nests are parallelized using the OpenMP `collapse` clause. Such loop nests are automatically transformed into collapsed loops that include the recovery of the original indices values, whose cost is minimized as described in Section V. Ranking polynomials and lexicographic minimums are computed using the ISL library [9], while polynomial equations are solved using the symbolic calculator Maxima.

Non-rectangular loop nests often occur in optimized code, where loop transformations as loop shifting, loop fission, skewing or tiling have been applied. Load imbalance and reduced parallelism often harms the expected benefit of such optimizations. Thus, we applied our tool to collapse loops that have previously been transformed into non-rectangular loops by the source-to-source loop optimizing and parallelizing compiler Pluto [10]. Some programs have also been transformed by tiling the loops (using flag `--tile` of Pluto), since tiling often yields incomplete tiles that affect load balancing. Pluto automatically inserts OpenMP loop parallelization pragmas before the outermost parallel loops (using flag `--parallel` of Pluto). We systematically tried to improve Pluto's parallelization by adding a `collapse` clause to the OpenMP pragma, whenever allowed by the dependences, in order to take advantage of enhanced parallelism and load balancing.

The target programs are 9 programs with kernel loop nests that have been extracted from the Polybench benchmark suite [11] and run using the `EXTRALARGE` dataset sizes. We also added to this set a loop nest program that computes the sum of two upper triangular $5,000 \times 5,000$ matrices (utma), and another one computing the product of two lower triangular $4,000 \times 4,000$ matrices (ltmp).

Programs have been compiled using gcc 5.4.0 with flags `-O3 -march=native -fopenmp -lm` and run on a 12-core AMD Opteron 6172 with 12 threads, on a machine using Linux 4.4.0-36. The reported execution-times are averages of five runs, and outputs of collapsed and non-collapsed programs have been compared to ensure the correctness of the collapsed loops.

The outermost loop of each original non-collapsed loop nest has been parallelized with OpenMP, either by using option `static` or `dynamic` of the `schedule` clause. For each program, the execution time of the most time-consuming non-rectangular loop nest has been measured. On Figure 9, we show the gains obtained by collapsing these loop nests and parallelizing them using option `static`. This gain has been calculated as follows:

$$gain = \frac{exec\_time\_without\_collapsing - exec\_time\_with\_collapsing}{exec\_time\_without\_collapsing}$$

The gains are obviously quite significant when comparing our collapsed loops to the original loops parallelized using option `static` (blue bars). They also often outperform the loops parallelized using option `dynamic`, or result in very close execution times (for correlation_tiled and covariance_tiled). For ltmp, option `dynamic` performs significantly better. This is because in the original 3-depth loop nest, only the two outermost loops could be collapsed, due to a data dependence carried by the innermost loop. Since this innermost loop has non-constant loop bounds (`for(k=j;k<i;k++)`), the resulting collapsed loop nest still exhibits a penalizing load imbalance among the 12 threads.

We also evaluated the time-overhead generated by the costly index recovery computation in the following way: we compared the serial execution times of the target loop nests (1) by running the original program without collapsing loops, and (2) by running the transformed program with collapsed loops, where root evaluations are performed 12 times, in order to simulate the computations performed with 12 threads. Time-overheads in percentages are represented

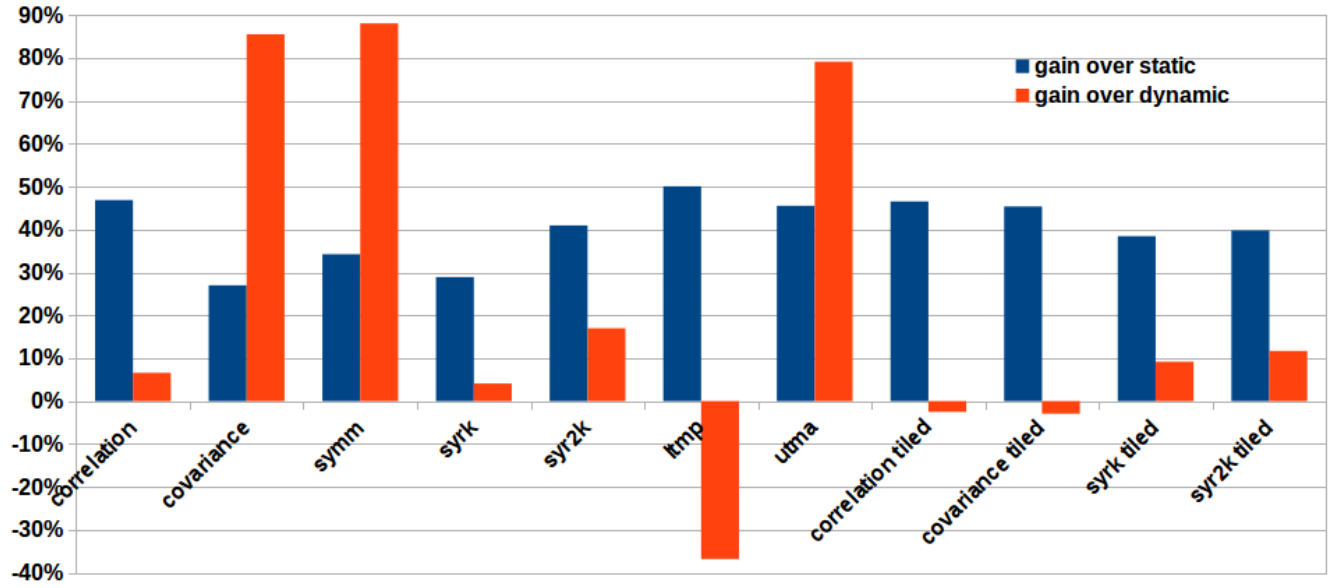

Figure 9. Gains on OpenMP execution times of collapsed non-rectangular loop nests (12 threads)

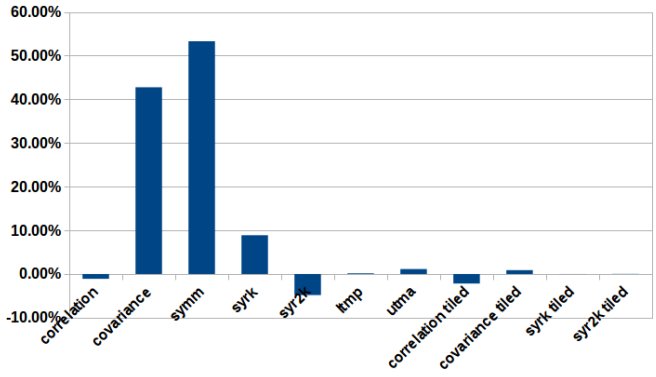

Figure 10. Control time-overhead from 12 root evaluations by comparing serial runs of original and transformed programs

in Figure 10. It shows that they are mostly small and even negligible, except when the collapsed loops are innermost, or when all the loops of the target loop nest have been collapsed (for covariance and symm). However, even in such latter cases, the time-overhead is greatly outweighed by the gains.

VIII. Related work

Loop collapsing was originally introduced by Polychronopoulos as loop coalescing [1], limited to perfectly nested loops with constant loop bounds, as it is currently implemented in OpenMP. Although there is no work that directly addresses collapsing of non-rectangular loops, some work propose either some dynamic scheduling of loop iterations [12], [13], or loop partitioning strategies [14], [15], [16], for well-balanced parallel multi-thread computations.

In [12], programmers must use OpenMP directives just to specify parallelism, annotating all application parallelism, and give the runtime library the responsibility of selecting the best way to exploit the available nested parallelism, using a function of the application characteristics and resource availability. In [13], the authors propose a system that allows a code to make a dynamic choice, at runtime, of what parallelism is applied to nested loops. The system works using a source to source compiler to perform transformations to user's code automatically, through a directive based approach that is similar to OpenMP. This approach requires the programmer to specify how the loops of the region can be parallelized and the runtime library is responsible for making the decisions dynamically during the execution of the code.

Sakellariou in [14] proposes a compile-time scheme for partitioning non-rectangular loop nests, where the minimization of load imbalance is based on symbolic cost estimates. In [15], the authors present a geometric approach for partitioning N-dimensional non-rectangular iteration spaces. They partition an iteration space along the axis corresponding to the outermost loop to achieve a near-optimal partition. The work presented in [16] focuses on static decomposition of perfect triangular iteration spaces to achieve load balancing, by partitioning a triangular iteration space of a loop nest along the outermost loop index.

IX. Conclusion

The presented technique, based on the inversion of ranking polynomials, allows to collapse any non-rectangular loop nest whose bounds are linear combinations of the loop iterators. We have shown that even if complex and

floating-point operations are required for initial recoveries of the original indices, the related time cost is quite small. Significant speed-ups are obtained, even when comparing the parallel collapsed loops with dynamically-scheduled parallel loops. Finally, the technique can be implemented to fully automatize the collapsing of non-rectangular loops.

We plan to extend our approach in the near future to imperfectly nested loops carrying data dependences. Other applications will also be investigated, as the computation of a loop nest from another loop nest of a different shape, or the fusion of loop nests of different shapes.

ACKNOWLEDGMENT

We would like to thank the reviewers for their valuable comments helping us improving the paper.

REFERENCES

[1] C. D. Polychronopoulos, "Loop coalescing: a compiler transformation for parallel machines," University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Technical Report CSRD-635, 1987.

[2] A. M. Ghuloum and A. L. Fisher, "Flattening and parallelizing irregular, recurrent loop nests," *SIGPLAN Not.*, vol. 30, no. 8, pp. 58–67, Aug. 1995.

[3] OpenMP, "OpenMP Application Program Interface Version 4.0," 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[4] D. L. Kreher and D. R. Stinson, *Combinatorial Algorithms: generation, enumeration, and search*. CRC Press, 1999.

[5] P. Clauss, "Counting Solutions to Linear and Nonlinear Constraints Through Ehrhart Polynomials: Applications to Analyze and Transform Scientific Programs," in *Proceedings of the 10th International Conference on Supercomputing*, ser. ICS '96, New York, NY, USA, 1996, pp. 278–285.

[6] P. Clauss and V. Loechner, "Parametric analysis of polyhedral iteration spaces," *J. VLSI Signal Process. Syst.*, vol. 19, no. 2, pp. 179–194, Jul. 1998.

[7] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe, "Counting integer points in parametric polytopes using barvinok's rational functions," *Algorithmica*, vol. 48, no. 1, pp. 37–66, 2007.

[8] P. Clauss and B. Meister, "Automatic memory layout transformations to optimize spatial locality in parameterized loop nests," *SIGARCH Comput. Archit. News*, vol. 28, no. 1, pp. 11–19, Mar. 2000.

[9] S. Verdoolaege, "ISL: An integer set library for the polyhedral model," in *Proc. of the Third Int. Conf. on Math. Software*, ser. LNCS 6327, 2010, pp. 299–302.

[10] "PLUTO - An automatic parallelizer and locality optimizer for multicores," http://pluto-compiler.sourceforge.net.

[11] "The Polyhedral Benchmark suite," http://sourceforge.net/projects/polybench.

[12] A. Duran, R. Silvera, J. Corbalán, and J. Labarta, "Runtime adjustment of parallel nested loops," in *Shared Memory Parallel Programming with Open MP: 5th International Workshop on Open MP Applications and Tools, WOMPAT 2004, Houston, TX, USA, May 17-18, 2004*, B. M. Chapman, Ed. Springer Berlin Heidelberg, 2005, pp. 137–147.

[13] A. Jackson and O. Agathokleous, "Dynamic loop parallelisation," *CoRR*, vol. abs/1205.2367, 2012.

[14] R. Sakellariou, "A compile-time partitioning strategy for non-rectangular loop nests," in *Proceedings of the 11th International Symposium on Parallel Processing*, ser. IPPS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 633–637.

[15] A. Kejariwal, P. Alberto, A. Nicolau, and C. D. Polychronopoulos, "A geometric approach for partitioning n-dimensional non-rectangular iteration spaces," in *Proceedings of the 17th International Conference on Languages and Compilers for High Performance Computing*, ser. LCPC'04. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 102–116.

[16] N. Kafri and J. A. Sbeih, "Simple near optimal partitioning approach to perfect triangular iteration space," in *Proceedings of the 2008 High Performance Computing & Simulation Conference*, 2008.