REGIONALES RECHENZENTRUM ERLANGEN
Friedrich-Alexander-Universität Erlangen-Nürnberg
Martenstr. 1, 91058 Erlangen, Germany

Basic Optimisation Strategies for CFD-Codes

Frank Deserno

11th July 2003

Official contact
Regionales Rechenzentrum Erlangen – RRZE
Martensstr. 1 • 91058 Erlangen, Germany
Tel.: +49 (0)9131 852 7031 • Fax.: +49 (0)9131 302941
http://www.rrze.uni-erlangen.de

High Performance Computing- (HPC-) Group
Dr. Gerhard Wellein
Tel.: +49 (0)9131 852 8737
e-mail: gerhard.wellein@rrze.uni-erlangen.de
http://www.rrze.uni-erlangen.de/dienste/hpc

©RRZE
This document was set using LaTeX.

Contents

1 SIP-Solver **3**
1.1 Introduction . . . 3
1.2 Incomplete LU-decomposition – SIP-solver . . . 4
1.3 Implementation . . . 5
1.3.1 Mapping of Indices . . . 6
1.3.2 3D-Version . . . 7
1.3.3 Hyperplane-Version . . . 9
1.3.4 Hyperline-Version . . . 13
1.3.5 SR8000-optimised version . . . 16
1.3.6 Pipeline-Parallel-Processing . . . 17
1.4 Results – Comparison . . . 19
1.4.1 Benchmark code . . . 19
1.4.2 Results . . . 20
1.4.3 Summary . . . 24

A Appendix **27**
A.1 SIP-Solver Benchmark . . . 27

1 SIP-Solver

In the following chapter a strongly-implicit-procedure (SIP-Solver) according to Stone is introduced [1]. This method is suitable for solving systems of linear equations resulting from a discretisation of partial differential equations (PDEs). It is a widely used method in fluid mechanics and therefore of great importance. Regarding the implementation, different approaches are introduced and compared.

1.1 Introduction

A discretisation via the finite volume method often results in a set of linear equations which can be written in matrix form as

$$A\boldsymbol{x} = \boldsymbol{b}.$$

The coefficients of $\boldsymbol{x}$ represent the values to be calculated (i.e. temperatures, velocities, ...) while the coefficients of A and $\boldsymbol{b}$ are generated by the discretisation and the boundary values. From the point of view of a given node in the lattice neighbour points have compass coordinates (N, S, E, W, NW, NE, B, T,...). For every lattice point (node) there is one equation. Considering a simple 2D-problem, the system might look like

$$\begin{pmatrix} a_{\mathrm{P}}^1 & a_{\mathrm{N}}^1 & 0 & a_{\mathrm{E}}^1 & & & \mathbf{0} \\ a_{\mathrm{S}}^2 & a_{\mathrm{P}}^2 & a_{\mathrm{N}}^2 & . & a_{\mathrm{E}}^2 & & \\ 0 & a_{\mathrm{S}}^3 & a_{\mathrm{P}}^3 & a_{\mathrm{N}}^3 & . & a_{\mathrm{E}}^3 & \\ a_{\mathrm{W}}^4 & . & a_{\mathrm{S}}^4 & a_{\mathrm{P}}^4 & a_{\mathrm{N}}^4 & . & a_{\mathrm{E}}^4 \\ & . & . & . & . & 0 & \\ & & . & . & . & . & a_{\mathrm{N}}^{N-1} \\ \mathbf{0} & & & a_{\mathrm{W}}^N & 0 & a_{\mathrm{S}}^N & a_{\mathrm{P}}^N \end{pmatrix} \begin{pmatrix} x_{\mathrm{P}}^1 \\ x_{\mathrm{P}}^2 \\ . \\ . \\ . \\ . \\ x_{\mathrm{P}}^N \end{pmatrix} = \begin{pmatrix} b_{\mathrm{P}}^1 \\ b_{\mathrm{P}}^2 \\ . \\ . \\ . \\ . \\ b_{\mathrm{P}}^N \end{pmatrix}$$

In this particular case, the coefficients of matrix A are placed on diagonals. In order to receive a solution for $\boldsymbol{x}$ one can determine a lower-upper-decomposition (LU-decomposition) of matrix A.

$$A = LU$$

A Gauss elimination yields the coefficients of U while the coefficients of L are the factors of the elimination process itself. Now we can define

$$U\boldsymbol{x} = \boldsymbol{y}.$$

Multiplication by L on both sides gives

$$\begin{aligned} LU\boldsymbol{x} &= L\boldsymbol{y} \\ A\boldsymbol{x} &= \boldsymbol{b} \Longrightarrow L\boldsymbol{y} = \boldsymbol{b} \end{aligned}$$

The solution can now be obtained by calculating $\boldsymbol{y}$ in $L\boldsymbol{y} = \boldsymbol{b}$ (forward substitution) and afterwards $\boldsymbol{x}$ from $U\boldsymbol{x} = \boldsymbol{y}$ (backward substitution).

Solving a system like this is possible but often not desirable as computing times can be quite high.

1.2 Incomplete LU-decomposition – SIP-solver

Like before, we have a system of linear equations given by $A\boldsymbol{x} = \boldsymbol{b}$. Now, the matrix A is to be decomposed in the following way:

$$A \approx LU = M = A + N.$$

L and U are a lower and upper triangle matrix. A multiplication of L and U yields diagonals which are not part of matrix A. These are put in N. N could be interpreted as a kind of error of the decomposition. To keep the deviation small we require

$$(A + N)\boldsymbol{x} \approx \boldsymbol{b} \Longrightarrow N\boldsymbol{x} \approx 0.$$

The coefficients on these extra diagonals are approximated by their neighbour coefficients. At this point knowledge of the structure of the PDE is used. Assuming an elliptic PDE the solution can be expected to be smooth. Therefore, a linear approximation with a parameter α can be made to determine these coefficients. I.e.

$$x_{\mathrm{NW}} \approx \alpha(x_{\mathrm{W}} + x_{\mathrm{N}} - x_{\mathrm{P}})$$

It is now possible to receive a set of *implicit* rules for the calculation of the coefficients of L and U. At step n we have

$$A\boldsymbol{x}^n = \boldsymbol{b} - \boldsymbol{r}^n$$

with $\boldsymbol{r}^n$ as the residual. The deviation of $\boldsymbol{x}^n$ from the real solution $\boldsymbol{x}$ is called convergence error $\boldsymbol{\epsilon}^n$

$$\boldsymbol{\epsilon}^n = \boldsymbol{x} - \boldsymbol{x}^n.$$

Therefore one yields

$$A\boldsymbol{\epsilon}^n = A\boldsymbol{x} - A\boldsymbol{x}^n = \boldsymbol{b} - \boldsymbol{b} + \boldsymbol{r}^n = \boldsymbol{r}^n \Longrightarrow A\boldsymbol{\epsilon}^n = \boldsymbol{r}^n.$$

As the real $\boldsymbol{x}$ is unknown, we construct an iterative rule:

$$\begin{aligned} A(\boldsymbol{x}^{n+1} - \boldsymbol{x}^n) &= \boldsymbol{r}^n \\ A\Delta\boldsymbol{x} = LU\Delta\boldsymbol{x} &= \boldsymbol{r}^n \\ U\Delta\boldsymbol{x} &= L^{-1}\boldsymbol{r}^n = \boldsymbol{R}^n \end{aligned}$$

Having done the decomposition we are now able to calculate the residual $\boldsymbol{r}^n$, the vector $\boldsymbol{R}^n$ and finally $\Delta\boldsymbol{x}$. The algorithm can now be formulated as:

Incomplete LU-decomposition:

(1) Do incomplete LU decomposition : $A \approx LU$
$\downarrow$
(2) Calculate residual : $\boldsymbol{r}^n = \boldsymbol{b} - A\boldsymbol{x}^n$
$\downarrow$
(3) Calculate vector $\boldsymbol{R}^n$ (forward substitution) : $\boldsymbol{R}^n = L^{-1}\boldsymbol{r}^n$
$\downarrow$
(4) Calculate $\Delta\boldsymbol{x}$ (backward substitution) : $U\Delta\boldsymbol{x} = \boldsymbol{R}^n$

(5) Back to (2), until residual is small enough.

In step 2 of the algorithm L^{-1} will not be calculated directly. Instead, an implicit rule like

$$R^l = \left(\rho^l - L^l_{\mathrm{S}} R^{l-1} - L^l_{\mathrm{W}} R^{l-N_j}\right) / L^l_{\mathrm{P}}$$

can be formulated, where l denotes the location in a 1D-array (see next section). The same yields for the backward substitution:

$$\Delta x^l = R^l - U^l_{\mathrm{N}} \Delta x^{l+1} - U_{\mathrm{E}} \Delta x^{l+N_j}$$

This is why the method is called strongly implicit.

1.3 Implementation

Taking a specific example, we will now have a look on different types of implementations for the SIP-Solver. The language of choice is Fortran 90. All compiler directives in the following listings refer to the f90-compiler on the Hitachi SR8000 at LRZ in Munich .

The rules for a 3D LU-decomposition can be written as follows:

$$\begin{aligned}
L^l_{\mathrm{B}} &= A^l_{\mathrm{B}} \left[1 + \alpha \left(U^{l-N_iN_j}_{\mathrm{N}} + U^{l-N_iN_j}_{\mathrm{E}}\right)\right]^{-1} \\
L^l_{\mathrm{W}} &= A^l_{\mathrm{W}} \left[1 + \alpha \left(U^{l-1}_{\mathrm{N}} U^{l-1}_{\mathrm{T}}\right)\right]^{-1} \\
L^l_{\mathrm{S}} &= A^l_{\mathrm{S}} \left[1 + \alpha \left(U^{l-N_i}_{\mathrm{E}} U^{l-N_i}_{\mathrm{T}}\right)\right]^{-1} \\
L^l_{\mathrm{P}} &= \ldots A^l_{\mathrm{P}} \ldots L^l_{\mathrm{B}} \ldots U^{l-N_iN_j}_{\mathrm{E}} \\
U^l_{\mathrm{N}} &= \ldots L^l_{\mathrm{P}} \ldots \\
U^l_{\mathrm{T}} &= \ldots L^l_{\mathrm{P}} \ldots \\
U^l_{\mathrm{E}} &= \ldots L^l_{\mathrm{P}} \ldots
\end{aligned}$$

N_i and N_j is the total number of nodes in i- and j-direction respectively. α is a parameter which results from the approximation of the coefficients of N. For our purpose $\alpha = 0.92$ is a proper value. Obviously there are dependencies for the coefficients of L and U. Within one loop L_{B} is first written and then read again for the calculation of other coefficients (marked blue). Moreover, before a nodes with index l is calculated all nodes $l - N_iN_j$, $l - N_i$ and $l - 1$ have to be processed (marked red).

We use a 1D-array (index l) to store the nodes in order to get optimal memory access. Therefore, a mapping from 3D- to 1D-coordinates is necessary.

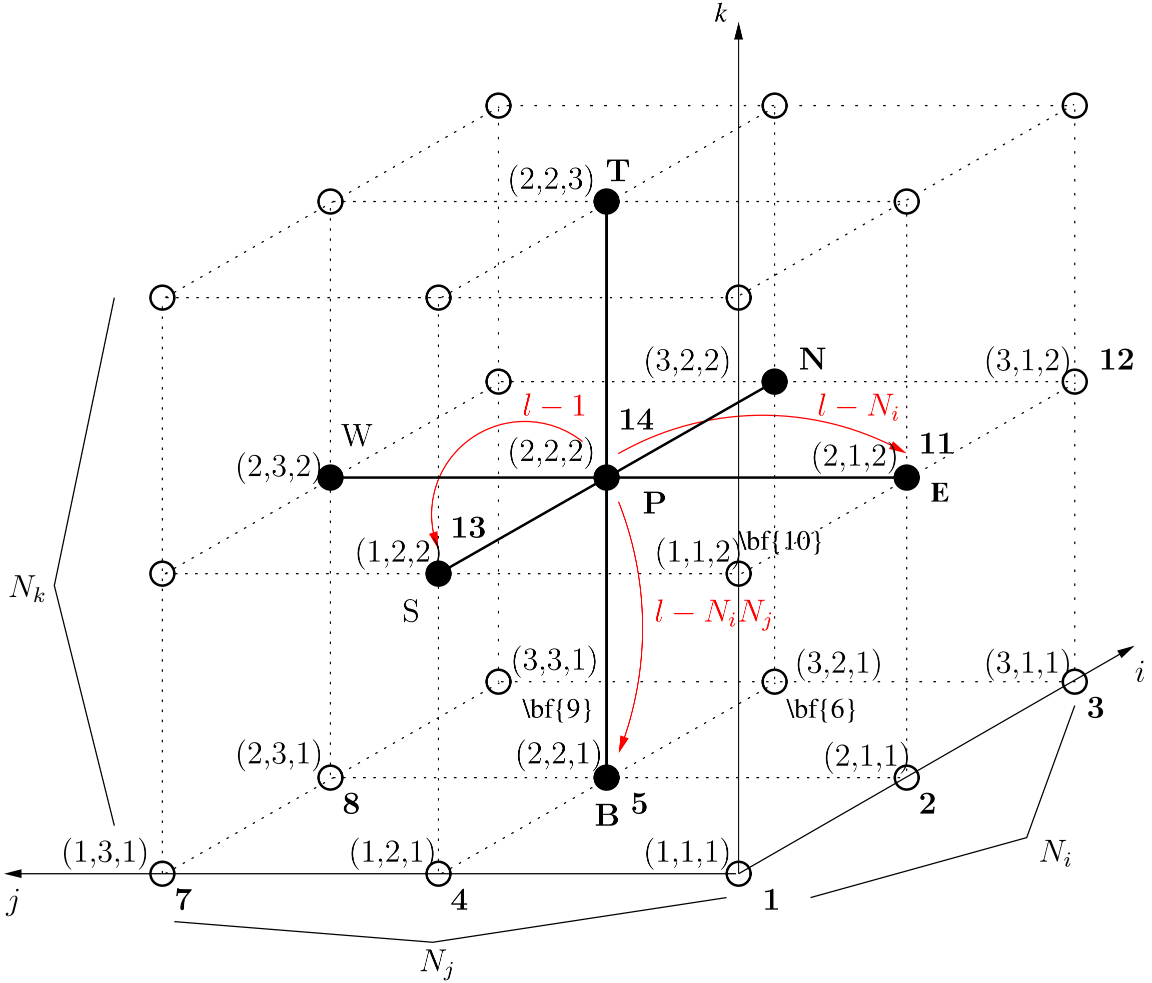

Figure 1.1: Numbering of nodes for the SIP-Solver in 3D. For all indices the rule $(i, j, k) \longrightarrow l = (k-1)N_iN_j + (j-1)N_i + i$ applies. In this picture $N_i = N_j = N_k = 3$. As an example, node $(2, 2, 2)$ maps to $l = (2-1) \cdot 3 \cdot 3 + (2-1) + \cdot 3 + 2 = 14$.

1.3.1 Mapping of Indices

According to the notation in the given rules each nodes (i, j, k) is mapped on a position l in a 1D-array. l can be calculated as

$$l = (k-1)N_iN_j + (j-1)N_i + i \tag{1.1}$$

Figure 1.1 shows the numbering of nodes in both cases.

Considering the rules and figure 1.1 we find that an access to node $l-1$ is an access to the previous node in i-direction. $l - N_i$ is an access to the node right of the reference node and access to $k-1$ would be an access on the node below.

Because of the dependencies in the LU-decomposition the rules cannot be computed independently for different nodes offhand, but have to be calculated in a strict order. As a result no parallelisation can be done at this point.

A closer look at the dependencies shows that there are only accesses to nodes $(i-1, j, k)$, $(i, j-1, k)$ or $(i, j, k-1)$ and thus accesses to $l - N_i$, $l-1$ and $l - N_iN_j$ respectively. In

order to achieve some independence of data one has to make sure that for a given node (i, j, k) the nodes $(i-1, j, k)$, $(i, j-1, k)$ and $(i, j, k-1)$ are calculated in advance. There are three different possibilities:

- Iteration over nodes (i, j, k) (3D-Version) without parallelisation.
- Iteration over nodes $i+k+j = \text{const.}$ (hyperplanes) with parallelisation within a hyperplane.
- Iteration over lines $j+k = \text{const.}$ (hyperlines) and parallel computation of j-lines within a hyperline.
- Pipeline parallel processing with parallelisation within blocks.

In the following we will discuss the different implementations.

1.3.2 3D-Version

A straight forward way is to iterate over all nodes (i, j, k) in 3 do-loops. As a preparation a conversion from 1D- to 3D-arrays has to be performed. This is easily possible using common blocks. In the main program 1D-arrays and 1D-indexing are used. For example:

```
1 PROGRAM SipMain
2
3 real*8 AE,AW,...
4 dimension AE(ijkMax),AW(ijkMax),...
5 common AE,AW
```

In the subroutine which contains the SIP-Solver the arrays are defined as 3D and put in common blocks again.

```
1 SUBROUTINE SipSolver
2
3 real*8 AE,AW,...
4 dimension AE(iMax,jMax,kMax),AW(iMax,jMax,kMax),...
5 common AE,AW
```

`iMax`, `jMax`, `kMax` denote the number of nodes in each dimension; `ijkMax` is the product of all three values. They are constant during compile time.

The coefficients of L and U are stored in a 3D-array, too. We name them `Lx` and `Ux` with `x` as one of the compass coordinates N, S, W, E, ... `FF` contains the values of the nodes which we like to determine. We use 3D-arrays for the coefficients of the system matrix as well and we call them `AB`, `AW`, `AS`, `AP`, `AN`, `AE` and `AT` (remember compass coordinates mentioned earlier). Now lets come to the algorithm itself.

LU-decomposition

A possible way to do a LU-decomposition is shown in the following listing:

```
1 do k=2,kMaxM
2    do j=2,jMaxM
3       do i=2,iMaxM
4          LB(i,j,k)=AB(i,j,k)/(1.+alpha*(UN(i,j,k-1)+UE(i,j,k-1)))
5          LW(i,j,k)=AW(i,j,k)/(1.+alpha*(UN(i-1,j,k)+UT(i-1,j,k)))
6          LS(i,j,k)=AS(i,j,k)/(1.+alpha*(UE(i,j-1,k)+UT(i,j-1,k)))
7
8          UN(i,j,k)=...
9          UE(i,j,k)=...
```

```
10          UT(i,j,k)=...
11       enddo
12    enddo
13 enddo
```

As a result of the data-dependencies and the conventional 3D-indexing no parallelisation is possible at first place. The compiler is able to apply prefetch instructions on the arrays `Ax` and `Ux`. Prefetch means that the compiler loads data from main memory into cache before he actually needs it in order to improve data throughput. Due to continuous memory access in this version prefetching results in a high number of cache hits. Application of those kind of instructions is visible in the compiler log-file.

The LU-decomposition has to be done only once during the whole process of acquiring the solution.

Residuals

The residuals are determined for each i-j-plane separately.

```
1  do k=2,kMaxM
2     do j=2,jMaxM
3        do i=2,iMaxM
4           RES(i,j,k)=Q(i,j,k)-AE(i,j,k)*FI(i+1,j,k)-AW(i,j,k)*
5   *          FI(i-1,j,k)-AN(i,j,k)*FI(i,j+1,k)-AS(i,j,k)*
6   *          FI(i,j-1,k)-AT(i,j,k)*FI(i,j,k+1)-AB(i,j,k)*
7   *          FI(i,j,k-1)-AP(i,j,k)*FI(i,j,k)
8
9           resN=resN+ABS(RES(i,j,k))
10        enddo
11    enddo
12 enddo
```

The array `RES` is used to store the values for the forward substitution (it is therefore overwritten). Note that the inversion of the matrix and multiplication with the vector is done in one step. Again no parallelisation is possible but prefetching results in a performance gain.

Forward substitution

The forward substitution can be done within the residual loop also. Here it is held separate[1]:

```
1  do k=2,kMaxM
2     do j=2,jMaxM
3        do i=2,iMaxM
4          RES(i,j,k)=(RES(i,j,k)-LB(i,j,k)*RES(i,j,k-1)-
5   *          LW(i,j,k)*RES(i-1,j,k)-LS(i,j,k)*RES(i,j-1,k))*
6   *          LP(i,j,k)
7        enddo
8     enddo
9  enddo
```

Backward substitution and value update

The last step of one iteration is the backward substitution and value update.

```
1  do k=kMaxM,2,-1
2     do j=jMaxM,2,-1
3        do i=iMaxM,2,-1
4           RES(i,j,k)=RES(i,j,k)-UN(i,j,k)*RES(i,j+1,k)-
5   *          UE(i,j,k)*RES(i+1,j,k)-UT(i,j,k)*RES(i,j,k+1)
```

[1] On the Hitachi SR8000 a separate loop results in a gain in performance. On other architectures the opposite was observed.

```
6
7           FI(i,j,k)=FI(i,j,k)+RES(i,j,k)
8         enddo
9       enddo
10    enddo
```

RES is overwritten in this part of the solver again. The last two steps have to be performed for every time step until the overall residual[2] is small enough. Again, putting the value update in an extra loop might result in a slight gain in performance.

1.3.3 Hyperplane-Version

A hyperplane is defined as

$$L = i + j + k = \text{const.}$$

Considering the rules for the LU-decomposition, for a given node (i, j, k) in hyperplane L only data of nodes of hyperplane $L-1$ are required because there is only one coordinate changing at a time (access on $(i-1, j, k)$, $(i, j-1, k)$ and $(i, j, k-1)$). Therefore, the calculation of nodes within a plane are independent of each other and can be parallelised. However, as a result of the dependency of L on $L-1$, hyperplanes have to be calculated one after another. Figure 1.2 shows hyperplanes in a cube.

In order to be able to do the iteration over hyperplanes one has to do some preparations. The number of hyperplanes in the system, the number of nodes per hyperplane and the indices of the nodes sorted by hyperplanes are the most important things to determine. We assume that we already have the following data pre-calculated:

`ijMax`	: number of nodes in a i-j-plane
`ijkMax`	: number of nodes in the whole domain
`LK(k)`	: number of nodes in all planes k = const. below the k-th plane
`LJ(j)`	: number of nodes in a plane k = const., in all lines j = const. below the current one (j-th)
`hyperplanes`	: number of hyperplanes
`LM(l)`	: number of nodes for hyperplane L
`ICL(l)`	: number of nodes in all planes before the L-th
`IJKV(LM(n)+ICL(l))`	: index of the n-th node in the l-th hyperplane, sorted by hyperplanes

The nodes themselves are stored in a 1D-array (conventional memory layout) and are referred to by the indices stored in IJKV. The first three hyperplanes in figure 1.2 contain 1, 3 and 6 nodes respectively. Using the indices-mapping already mentioned array IJKV would look like this:

$$\texttt{IJKV} = [\underbrace{1,}_{L=1}\ \underbrace{2, 6, 26,}_{L=2}\underbrace{3, 7, 11, 27, 31, 51,}_{L=3} \ldots]$$

The coefficients of L and U are stored in a 1D-array, too. Again, we name them Lx and Ux. FF contains the values of the nodes which we like to determine. We use 1D-arrays for the coefficients of the system matrix as well and we call them Ax.

LU-decomposition

A possible way to do a LU-decomposition in 1D-indexing is shown in the following listing:

[2] In this case the residual is the sum all residuals of all nodes.

Figure 1.2: Hyperplanes in a cube. For the sake of clarity only every second plane is drawn. The filled circles show the nodes for the first three hyperplanes (the first is just one node). Bold numbers mark the nodes in 1D-notation.

```
1  do l=1,hyperplanes
2
3     n=ICL(l)
4
5  !$omp parallel do private(ijk,p1,p2,p3)
6  *voption indep
7  *poption parallel
8
9     do m=1,LM(l)
10
11       ijk=IJKV(m+n)
12
13       LB(ijk)=AB(ijk)/(1.+alpha*(UN(ijk-ijMax)+UE(ijk-ijMax)))
14       LW(ijk)=AW(ijk)/(1.+alpha*(UN(ijk-1)+UT(ijk-1)))
15       LS(ijk)=AS(ijk)/(1.+alpha*(UE(ijk-iMax)+UT(ijk-iMax)))
16       ...
17    enddo
18 enddo
```

We iterate over all hyperplanes in the outer loop and over all nodes in a plane in the inner loop. The directives in line 6 and 7 tell the compiler to assume independency of the instructions for nodes within a plane (f90-compiler at Hitachi SR8000 at LRZ Munich). OpenMP-parallelisation could also be used. `ijk` is the index of the particular node. Because we do

not have a continous memory access for the arrays the compiler tries to apply preload to the arrays Ax, Lx and Ux to hide the latency from memory access. Preload means that values from main memory are loaded into the registers directly. The cache is not involved in this process. Application of preload instructions can be seen in the compiler logfile. In this particular case the compiler changes from preload to prefetch due to a lack of registers (no software pipelining possible). To assist the compiler we can change the inner loop a bit. We take the loop variable m as index for LB, LW and LS. All other arrays are left untouched.

```
1  do m=n+1,n+LM(l)
2
3     ijk=IJKV(m)
4
5     LB(m)=AB(ijk)/(1.+alpha*(UN(ijk-ijMax)+UE(ijk-ijMax)))
6     LW(m)=AW(ijk)/(1.+alpha*(UN(ijk-1)+UT(ijk-1)))
7     LS(m)=AS(ijk)/(1.+alpha*(UE(ijk-iMax)+UT(ijk-iMax)))
8     ...
9  enddo
```

Now we have continuous memory access for Lx and the compiler is able to do prefetch instead of preload. As a result, no register spill is observed and software pipelining can be applied.

Residuals

The following listing shows the calculation of the residuals for the hyperplane version in a time step. Here we have to split the calculation of residuals and the forward substitution for the sake of parallelisation.

```
1  do k=2,kMaxM
2
3     lkk=LK(k)
4
5  !$omp parallel do reduction (+:resN) private(ijk)
6  *voption indep
7  *poption parallel
8
9     do ij=jMax+2,ijMax-jMax-1
10
11       ijk=lkk+ij
12
13       RES(ijk)=QE(ijk)-AP(ijk)*FF(ijk)-
14  *       AE(ijk)*FF(ijk+1)-AW(ijk)*FF(ijk-1)-
15  *       AN(ijk)*FF(ijk+iMax)-AS(ijk)*FF(ijk-iMax)-
16  *       AT(ijk)*FF(ijk+ijMax)-AB(ijk)*FF(ijk-ijMax)
17
18       resN=resN+ABS(RES(ijk))
19
20    enddo
21 enddo
```

The residuals are calculated for one i-j-plane. This can be done in parallel within a plane as there are no data dependencies. All residuals are added up to one value (OpenMP reduction clause). Prefetch can be applied to arrays QE, FF and Ax.

Forward substitution

Forward substitution can be done as follows:

```
1  do l=1,hyperplanes
2
3     n=ICL(l)
```

```
4
5  !$omp parallel do private(ijk)
6  *voption indep
7  *poption parallel
8
9     do m=n+1,n+LM(l)
10
11       ijk=IJKV(m)
12
13       RES(ijk)=(RES(ijk)-LB(ijk)*RES(ijk-ijMax)-LW(ijk)*
14  *         RES(ijk-1)-LS(ijk)*RES(ijk-iMax))*LP(ijk)
15    enddo
16
17 enddo
```

We iterate over all hyperplanes in the outer loop. The inner loop (line 9) starts from the first node in the particular hyperplane and stops at the last one. RES is overwritten during this process. By telling the compiler the right directive, parallelisation is again possible within a hyperplane. As all indices of nodes in the array IJKV are in order, consecutive memory access and therefore prefetch is possible. On RES and Lx preload is applied. If the LU-decomposition was changed in the proposed manner to prevent register spill, we can try to change indexing for LB, LW and LS accordingly.

```
1  do m=n+1,n+LM(l)
2
3     ijk=IJKV(m)
4
5     RES(ijk)=(RES(ijk)-LB(m)*RES(ijk-ijMax)-LW(m)*
6   *       RES(ijk-1)-LS(m)*RES(ijk-iMax))*LP(m)
7
8  enddo
```

Now prefetch instead of preload for Lx is possible which results in a performance gain for this routine (943 compared to 418 MFlops/s for a system of 91^3).

Backward substitution and value update

Backward substitution and value update is again done by an iteration over all hyperplanes with parallelisation within a plane. This is similar to the 3D-version except for the array indexing.

```
1  do l=hyperplanes,1,-1
2
3     n=ICL(L)
4
5  !$omp parallel do private(ijk)
6  *voption indep
7  *poption parallel
8
9     do m=n+1,n+LM(l)
10
11       ijk=IJKV(m)
12
13       RES(ijk)=RES(ijk)-UN(ijk)*RES(ijk+iMax)-UE(ijk)*
14  *         RES(ijk+1)-UT(ijk)*RES(ijk+ijMax)
15
16       FF(ijk)=FF(ijk)+RES(ijk)
17    enddo
18
19 enddo
```

The update of the values can be done in an extra loop also. The last 3 steps have to be performed over and over again until the residual is small enough. Preload and prefetch is done in the same way as in the forward substitution.

1.3.4 Hyperline-Version

Instead of hyperplanes one can easily define hyperlines like

$$L = j + k = \text{const.}$$

Similar to the hyperplane-version there is no need for a node in one diagonal to access another node in the same diagonal. Therefore, within a hyperline the lines in i-direction can be processed in parallel. This is shown in figure 1.3.

Figure 1.3: Hyperlines in a cube. The hyperline containing node 6 and 26 shows the idea of parallelisation. Each row in j-direction can be calculated independently. Bold numbers mark the nodes in 1D-notation.

As for the hyperplanes some pre-calculations have to be done.

`ijMax`	: number of nodes in a i-j-plane
`ijkMax`	: number of nodes in the whole domain
`LK(k)`	: number of nodes in all planes $k = \text{const.}$ below the k-th plane

`LJ(j)`	: number of nodes in a plane k = const., in all lines j = const. below the current one (j-th)
`hyperlines`	: number of hyperlines
`lenDiag(l)`	: number of nodes in a hyperline
`startStopDiag(1|2,l)`	: pointer to the first and last node in hyperline l
`coordDiag(1|2,n)`	: $k\|j$ coordinate of node n

The different parts of the algorithm look similar to the previous ones.

LU-decomposition

LU-decomposition can be implemented in the following way:

```
1  do l=1,hyperlines
2
3    if(lenDiag(l) .gt. 0) then
4
5  !$omp parallel do private(k,i,j,ijk,p1,p2,p3)
6  *voption indep
7  *poption parallel
8
9      do diag = startStopDiag(1,l) , startStopDiag(2,l)
10
11        k=coordDiag(1,diag)
12        j=coordDiag(2,diag)
13
14        do j=2,jMaxM
15           ijk=(k-1)*ijMax+(j-1)*iMax+i
16
17           LB(ijk)=EB(ijk)/(1.+alpha*(UN(ijk-ijMax)+
18   *          UE(ijk-jMaxM)))
19           LW(ijk)=EW(ijk)/(1.+alpha*(UN(ijk-1)+UT(ijk-1)))
20           LS(ijk)=ES(ijk)/(1.+alpha*(UE(ijk-jMax)+UT(ijk-jMax)))
21           ...
22        enddo
23     enddo
24   endif
25 enddo
```

We iterate over all hyperlines. If there is at least one node in the line, we process all nodes in lines in i-direction in parallel (compiler directives in lines 6 and 7, see figure 1.3). Adresses of nodes are detemined in line 15 according to relation (1.1). As in the 3D-version the compiler can apply prefetch to arrays `Ex` and `Ux` as all nodes in i-direction are calculated one after another.

Residuals

The following listing shows the calculation of the residuals for the hyperplane version in a time step. We change the indexing here from 1D to 3D but still proceed along hyperlines. This leads to much better performance on cache based machines.

```
1  !$omp parallel do reduction (+:resN) private(i,j,ijk)
2  *voption indep
3  *poption parallel
4
5  do k=2,kMaxM
6    do j=2,jMaxM
7      do i=2,iMaxM
8
```

```
 9          RES(i,j,k)=QE(i,j,k)-EP(i,j,k)*FF(i,j,k)-
10   *         EE(i,j,k)*FF(i+1,j,k)-EW(i,j,k)*FF(i-1,j,k)-
11   *         EN(i,j,k)*FF(i,j+1,k)-ES(i,j,k)*FF(i,j-1,k)-
12   *         ET(i,j,k)*FF(i,j,k+1)-EB(i,j,k)*FF(i,j,k-1)
13
14          resN = resN+ABS(RES(i,j,k))
15        enddo
16      enddo
17    enddo
```

Like the hyperplane version calculating residuals implies no data dependencies. We run over successive nodes in a i-j-plane and can therefore apply prefetch-operations.

Forward substitution

Again we iterate over all hyperlines with parallelisation of all lines in i-directions within one diagonal.

```
 1 do l=1,hyperlines
 2
 3   if(lenDiag(l) .gt. 0) then
 4
 5 !$omp parallel do private(k,i,j,ijk)
 6
 7     do diag = startStopDiag(1,l) , startStopDiag(2,l)
 8
 9       k=coordDiag(1,diag)
10       j=coordDiag(2,diag)
11
12       do i=2,iMaxM
13
14         RES(i,j,k)=(RES(i,j,k)-LB(i,j,k)*RES(i,j,k-1)-LW(i,j,k)*
15  *          RES(i-1,j,k)-LS(i,j,k)*RES(i,j-1,k))*LP(i,j,k)
16       enddo
17     enddo
18   endif
19 enddo
```

Prefetch is possible for arrays RES and Lx.

Backward substitution and value update

```
 1 do l=hyperlines,1,-1
 2
 3   if(lenDiag(l) .gt. 0) then
 4
 5 !$omp parallel do private(k,i,j,ijk)
 6
 7     do diag = startStopDiag(1,l) , startStopDiag(2,l)
 8
 9       k=coordDiag(1,diag)
10       j=coordDiag(2,diag)
11
12       do i=iMaxM,2,-1
13
14         RES(i,j,k)=RES(i,j,k)-UN(i,j,k)*RES(i,j+1,k)-UE(i,j,k)*
15  *          RES(i+1,j,k)-UT(i,j,k)*RES(i,j,k+1)
16
17         FF(i,j,k)=FF(i,j,k)+RES(i,j,k)
18       enddo
19     enddo
```

```
20      endif
21   enddo
```

As with the hyperplane version the last 3 steps have to be performed until the residual is small enough.

1.3.5 SR8000-optimised version

For the Hitachi SR8000-F1 at the LRZ some modifications turned out to further improve the performance. As a basis the hyperplane-version was taken. The LU-decomposition is left untouched but at the forward-substitution one has to put in a `soption nounroll`-directive which leads to a higher level of unrolling than before (24 compared to 16). The compiler-log without the option is given here:

```
1  *voption indep
2  ...
3  **    Innermost loop unrolled (2 times).
4  **    Stride pattern analyzed, 5 streams (LB[], LP[], LS[],
5  **    LW[], VEC.IJKV[][]) pre-fetch assumed.
6  **    Stride pattern analyzed, 6 streams (RES[]x6) pre-load
7  **    assumed.
8  **    With assumed latency=21,3 stages of SWPL, slide width
9  **    8 applied.
10 **    6 streams (RES[]x6) pre-load applied.
11 ...
12 **    Estimated logical performance:
13 **             176.47( 308.82) Mflops at 375MHz/PE
14 ...
15 **    Estimated instructions:
16 **             QLD FLD ILD QST FST IST +/- MPY MPY+/- FDIV/SQRT
17 **               6   4   2   0   2   0   0   2     6        0
18 **             FCONV FOP BR. OTHER
19 **                 0   0   1   5
20 **    Generated instructions:
21 **             QLD FLD ILD QST FST IST +/- MPY MPY+/- FDIV/SQRT
22 **              48  32  32   0  16   0   0  16    48        0
23 **             FCONV FOP BR. OTHER
24 **                 0   0   8  69
25 **    initiation intervall=17, With assumed latency=250,
26 **    loop unrolling 8 times
27 ...
```

The compiler applies 2 times loop-unrolling (line 3) and gets therefore $2 \cdot 3 = 6$ streams for `RES` for which it has to apply pre-load (scattered in memory). For each array `LB`, `LP`, `LS` and `LW` only one stream has to be created (linear access to memory, prefetch). At the end of the log-file the compiler gives the information that the loop has been unrolled 8 times more which leads to a level of $2 \cdot 8 = 16$ for unrolling. This can be seen in the difference between the estimated and generated instructions, too. The loop-unrolling by a factor of 2 is already incorporated in the estimated instructions.

However, a simple `soption nounroll`-directive changes a lot.

```
1  *voption indep
2  *soption nounroll
3  ...
4  **    Stride pattern analyzed, 5 streams (LB[], LP[], LS[],
5  **    LW[], VEC.IJKV[][]) pre-fetch assumed.
6  **    Stride pattern analyzed, 3 streams (RES[]x3) pre-load
7  **    assumed.
```

```
 8  **    With assumed latency=165,18 stages of SWPL, slide width
 9  **    4 applied,
10  **    6 times modulo variable expansion.
11  **    3 streams (RES[]x3) pre-load applied.
12  ...
13  **    Estimated logical performance:
14  **              107.14( 187.50) Mflops at 375MHz/PE
15  ...
16  **    Estimated instructions:
17  **              QLD FLD ILD QST FST IST +/- MPY MPY+/- FDIV/SQRT
18  **                1   6   1   0   1   0   0   1      3         0
19  **              FCONV FOP BR. OTHER
20  **                  0   0   1   4
21  **    Generated instructions:
22  **              QLD FLD ILD QST FST IST +/- MPY MPY+/- FDIV/SQRT
23  **               24 144  48   0  24   0   0  24     72         0
24  **              FCONV FOP BR. OTHER
25  **                  0   0   4 129
26  **    initiation interval=84, With assumed latency=250,
27  **    loop unrolling 4 times with exit
28  ...
```

Now we have loop-unrolling by a level of 4 and 18 instead of 3 stages of software-pipelining (SWPL). Moreover, we now get 6 times modulo variable expansion which results in $6 \cdot 4 = 24$ levels of unrolling (see again change from estimated to generated instructions).

So by adding a `soption nounroll`-directive we actually achieved an even higher level of unrolling. We will see that the latter version yields higher performance then the first one.

1.3.6 Pipeline-Parallel-Processing

As mentioned the result of the data dependencies prohibit parallelisation of the 3D-Version of the SIP-Solver. However, the Fortran90-compiler at the Hitachi is able to resolve the dependencies on its own be applying pipeline parallel processing. The system is divided up into chunks of a certain size according to figure 1.4. Parallelisation is done for the loop along the j-direction (the middle loop). Calculation of any chunk is delayed by a barrier until the chunk left of it is processed. Consequently, blocks with equal colour in figure 1.4 are calculated concurrently by different CPUs in a compute-node. This leads to load imbalance in the “windup” and ”wind-down” phases of this pipeline as some CPUs have to wait for the first few chunks to be calculated. This effect is negligible for a large enough lattice.

This kind of processing is very similar to the hyperline-version except for the size of the chunks (only lines for the hyperline-version). As indicated by the arrows inside the rightmost ‘C’ block in Fig. 1.4, outer loop unrolling is possible to reduce the load-to-flop ratio.

This kind of parallelisation can be also done by hand using OpenMP directives. A typical loop, i.e. the calculation of the residuals, looks like follows:

```
 1  !$omp parallel private(i,j,k,l,threadID,p1,p2,p3,resN,iter,resNor,rsm)
 2
 3  ...
 4
 5  do l=2,kMax+numThreads-2
 6
 7     threadID = OMP_GET_THREAD_NUM()
 8     k = l - threadID
 9
10     if ((k.ge.2).and.(k.le.kMaxM)) then
```

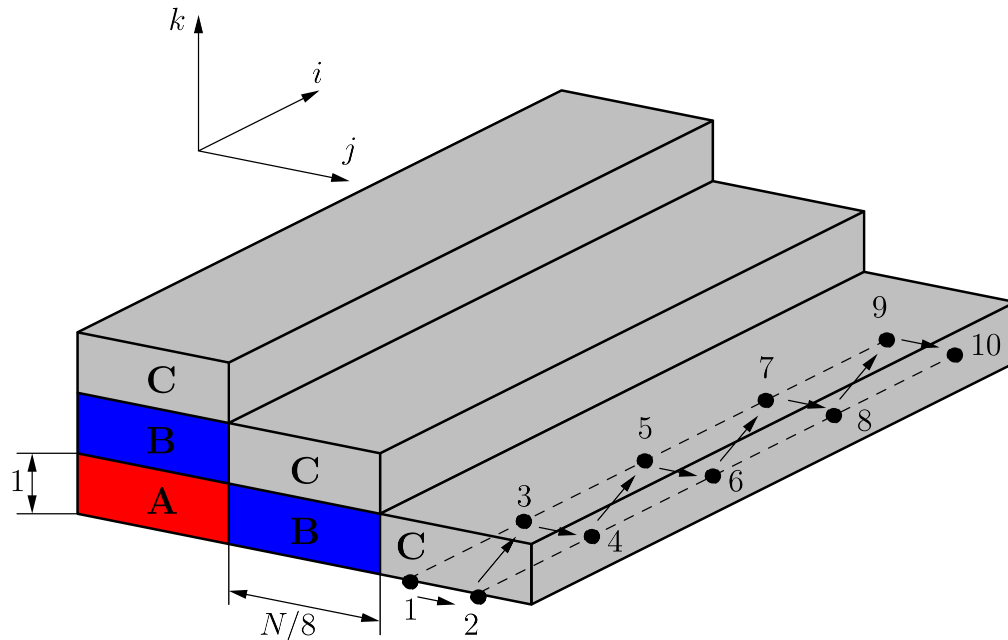

Figure 1.4: Schematic view of pipeline parallel processing. Blocks with equal colours (or letters) are calculated simultaneously. One CPU always processes the nodes inside a certain range of the middle loop (loop variable j).

```
11
12          do j=jStart(threadID),jEnd(threadID)
13             do i=2,iMaxM
14                RES(i,j,k)=Q(i,j,k)-AE(i,j,k)*FI(i+1,j,k)-AW(i,j,k)
15     *             *FI(i-1,j,k)-AN(i,j,k)*FI(i,j+1,k)-AS(i,j,k)*
16     *             FI(i,j-1,k)-AT(i,j,k)*FI(i,j,k+1)-AB(i,j,k)*
17     *             FI(i,j,k-1)-AP(i,j,k)*FI(i,j,k)
18
19                resN=resN+ABS(RES(i,j,k))
20             enddo
21          enddo
22
23      endif
24
25    !$omp barrier
26
27    enddo
28
29    ...
30
31    !$omp end parallel
```

The system is divided up in junks as depicted in figure 1.4. Before each threads starts calculating it has to make sure it is allowed to start with its junk (meaning all junks left of it are already calculated). If this is the case (line 10) the threads starts doing its junk. If not, it hits the barrier in line 25. Of course, the calculation of the junk-size has to be done in advance.

Table 1.1: SIP-solver benchmark: problem sizes, number of iterations and floating point operations.

problem size	# of iterations	# of MFlops
31^3	10000	7074
41^3	5000	8604
51^3	2500	8535
61^3	1500	8943
71^3	1000	9542
81^3	700	10031
91^3	500	10254

1.4 Results – Comparison

1.4.1 Benchmark code

The results presented in the following section were acquired by a self-made benchmark code[3]. It contains a variety of versions of the SIP-solver and measures wall clock time, cpu-time, performance and additionally times the different parts of each solver type. All output for one run is stored in a file. A perl script controls the whole experiment by setting proper links to make- and parameter include files, compiling the code and running it for several times. Finally it accumulates all results in a single output file (see appendix for detailed information about the benchmark code).

To achieve a proper run time problem sizes from 31^3 to 91^3 were used and the number of iterations were set to a appropriate value for each size. The number of floating point operations for a complete run for a specific size was pre-calculated and used to determine the performance. Table 1.1 shows corresponding numbers.

The code has been run on different architectures for now. For reasons of availability and interest two of these are of special interest, namely the SGI Origin 3400 at the RRZE and the Hitachi SR8000 at the LRZ. Some results for vector machines (NEC SX5 at the HLRS, Fujitsu VPP300 at the RRZE) and IBM Power/Intel Itanium will also be shown. Calculations were all done with double precision floating points values unless specified different.

As already mentioned the Hitachi compiler is able to resolve the data dependencies in the 3D-version, too. Unfortunately, the Fortran-compiler on the SGI is not able to do so. Some of the switches provide a lot of compiler output, which is quite useful. I.e., during compilation the log-file mentioned earlier (`file.log`) is produced. It contains valuable information of what the compiler actually did with the code (parallelisation, pseudo-vectorisation, application of prefetch-/preload instructions . . .).

The matrix of the linear system to be solved results from a 3D heat transfer problem. Boundary conditions are of Dirichlet type: $T(0, x, z) = T(x, 0, z) = T(x, y, 0) = 0$ and $T(1, y, z) = y \cdot z$, $T(x, 1, z) = x \cdot z$, $T(x, y, 1) = x \cdot y$ (x, y and z run from 0 to 1). The exact solution is $T(x, y, z) = x \cdot y \cdot z$. A result file for a system of 41^3 is shown in figure 1.5.

[3] The code can be received at the HPC-group of the RRZE. Please mail to hpc@rrze.uni-erlangen.de.

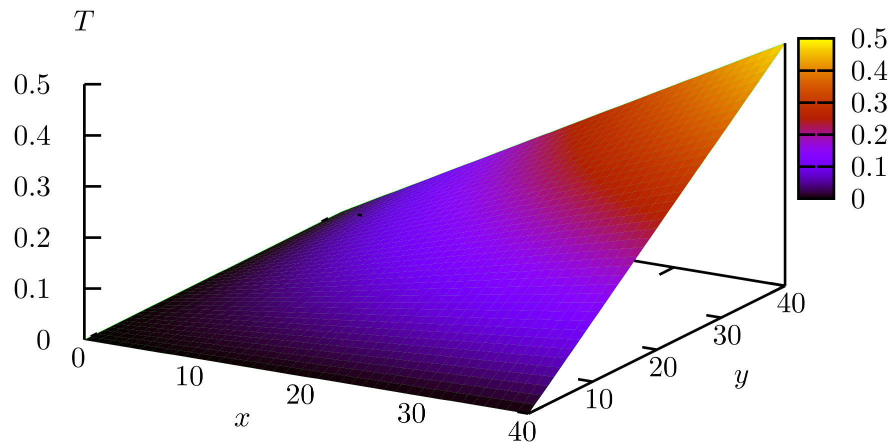

Figure 1.5: Visualisation of a result of a benchmark case (system size 41^3). The matrix of the linear system to be solved results from a 3D heat transfer problem. Boundary conditions are of Dirichlet type: $T(0, x, z) = T(x, 0, z) = T(x, y, 0) = 0$ and $T(1, y, z) = y \cdot z$, $T(x, 1, z) = x \cdot z$, $T(x, y, 1) = x \cdot y$ (x, y and z run from 0 to 1). The exact solution is $T(x, y, z) = x \cdot y \cdot z$.

1.4.2 Results

Figure 1.6 shows the performance on the SGI Origin 3400 for 1 and 8 CPUs for different problem sizes. The hyperplane-version performs worst, the 3D-version best. This is due to the fact that the latter version produces much more cache hits than the first one. Every access on a node (i, j, k) loads elements $(i, j + C, k)$ into cache as well. These nodes will be computed next so they will produce a cache hit. Accessing a node (i, j, k) in the hyperplane version also loads the elements $(i, j+C, k)$ into cache but unfortunately they will be used much later (they belong to a different hyperplane) and therefore will be most probably thrown out of cache before. The hyperline-version iterates over different j-lines and produces cache hits at least for those.

The option of partial parallelisation for the hyperplane- and hyperline-version yields the possibility to use 8 CPUs in these cases. Because of the increased number of cache hits for the hyperline version, it performs better on 8 CPUs than the hyperplane-version.

Figure 1.7 shows the performance on the Hitachi SR8000 for 1 and 8 CPUs on the IAPAR partition for different problem sizes. Similar to the SGI Origin for 1 CPU clearly the 3D-version with no parallelisation has the best performance. In this case the compiler is capable of parallelising the 3D-version and so an 8 CPU run is possible for this version, too. With 8 CPUs the hyperline- and the 3D-version perform equally well and the hyperplane-version is a bit slower (fewer cache hits).

Figure 1.8 shows the difference between single and double precision floating point numbers. The 3D-version with single precision performs remarkably better than the one with double precision values. This is due to the fact, that for `real*8` only half of the amount of floating point numbers fit into cache as for `real*4`. The same happens for the hyperplane-version.

With the Hitachi SR8000 there is no pronounced effect for the pipeline parallel version

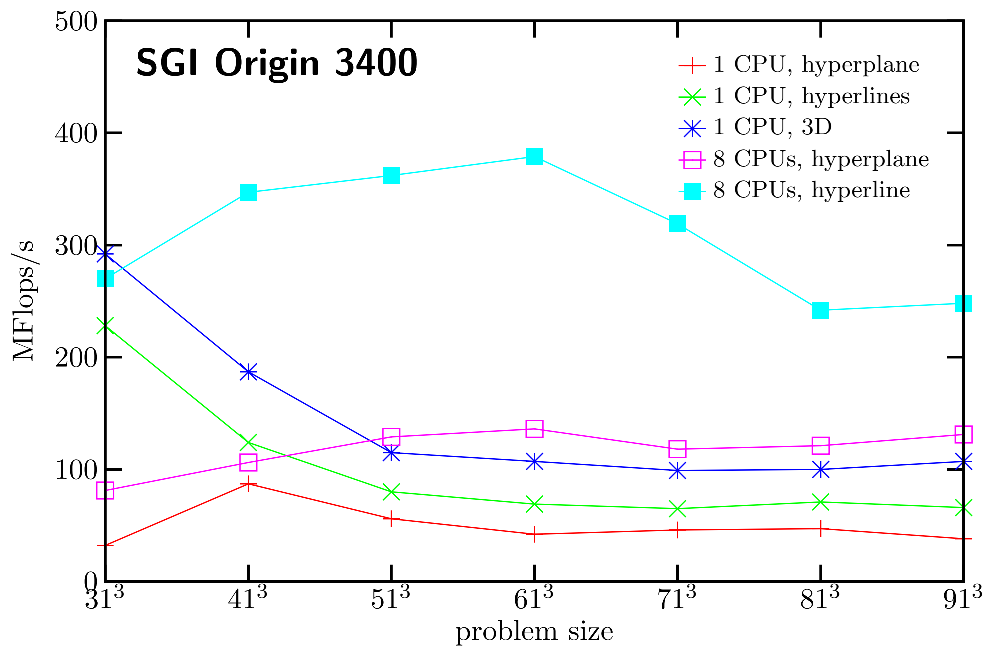

Figure 1.6: SGI Origin 3400: performance of all three versions of the SIP-solver for 1 and 8 CPUs respectively (double precision).

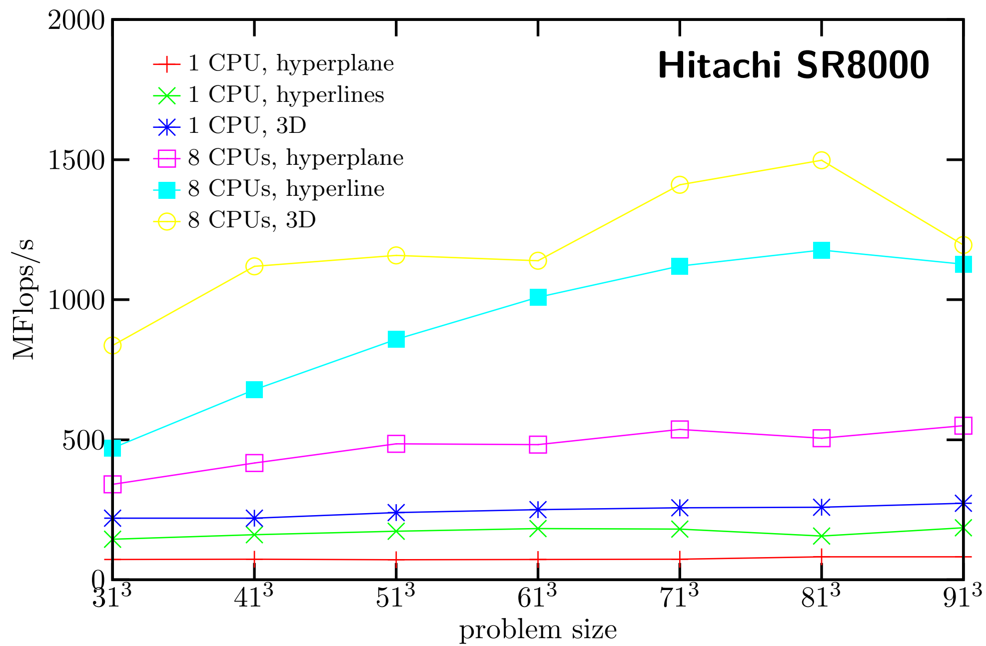

Figure 1.7: Hitachi SR 8000: performance of all three versions of the SIP-solver for 1 and 8 CPUs respectively (double precision).

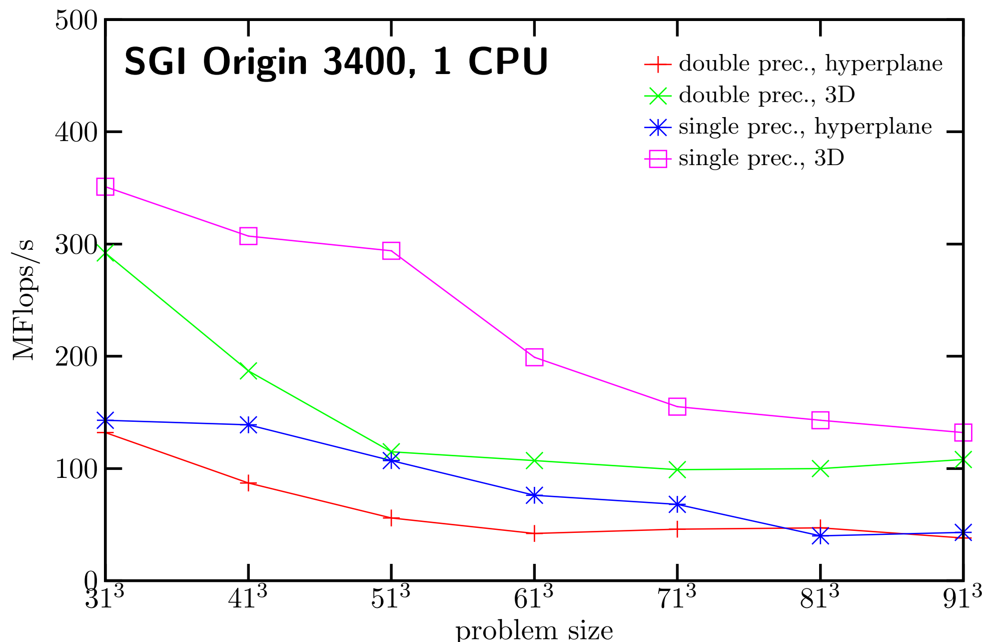

Figure 1.8: SGI Origin 3400: performance of SIP-solver variants. Influence of single/double precision.

(figure 1.9). However, the hyperplane version performs much better for double-precision values (real*8) than for single precision. As the SR8000 can do preload only for double precision values this fact is not astonishing because preloads are heavily used for the hyperplane-version. The difference in performance is much smaller for the 3D-version since in this case prefetch instead of preload is applied (continuous memory access).

Figure 1.10 contains performance data for different platforms including two vector machines. The hyperplane-version shows incredible performance on the NEC SX5 and Fujitsu VPP300 for a single CPU run. This version is highly adapted for those architectures as it provides long vector lengths within hyperplanes. Obviously there are periodic degradations of the performance for the vector machines. This has not been investigated further and could be due to disadvantageous memory access[4]. Nevertheless, the 3D version on one Hitachi node outperforms all other versions by far.

As described before, it is possible to receive a version optimised for the SR8000-F1. Its performance compared to the usual versions is shown in figure 1.11. One can see a slight improve with bigger systems. The MFlops/s-numbers for the different part are as follows (system size 91^3):

LU-decomposition	: 1567 MFlops/s
Residual	: 3923 MFlops/s
Forward substitution	: 1010 MFlops/s
Backward substituion	: 1067 MFlops/s

Figure 1.12 shows some results for the pipeline-parallel-version. The Hitachi SR8000-F1 shows a very good scaling and achieves 1698 MFlops/s compared to 1584 MFlops/s of the optimised version. On the SGI the ppp-version performs better than the hyperline-version.

[4]Actually, the system size is not a power of 2 and therefore it is not quite clear what is going on here.

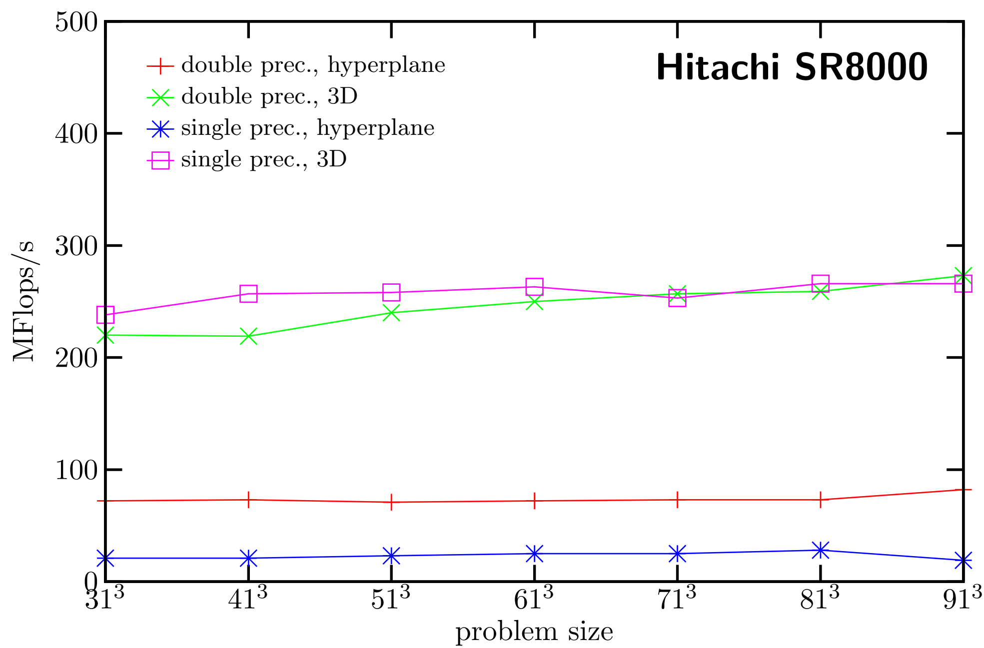

Figure 1.9: Hitachi: performance for SIP-solver variants. Influence of single/double precision.

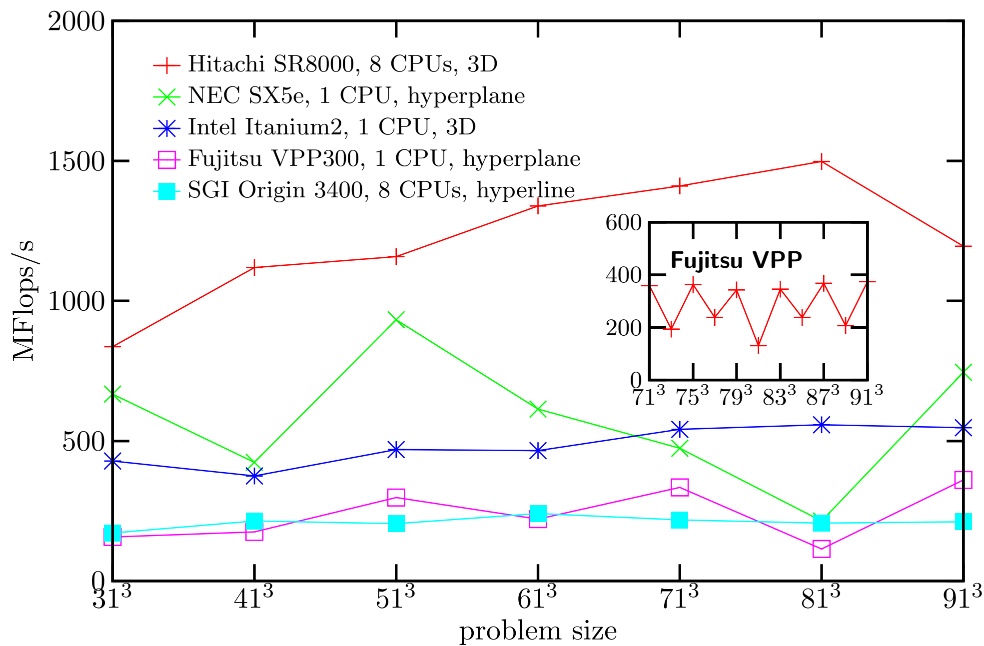

Figure 1.10: SIP-solver benchmark: performance on different architectures, considering the best version for each. The performance degradation for certain problem sizes has not been investigated further but could be due to disadvantageous memory access.

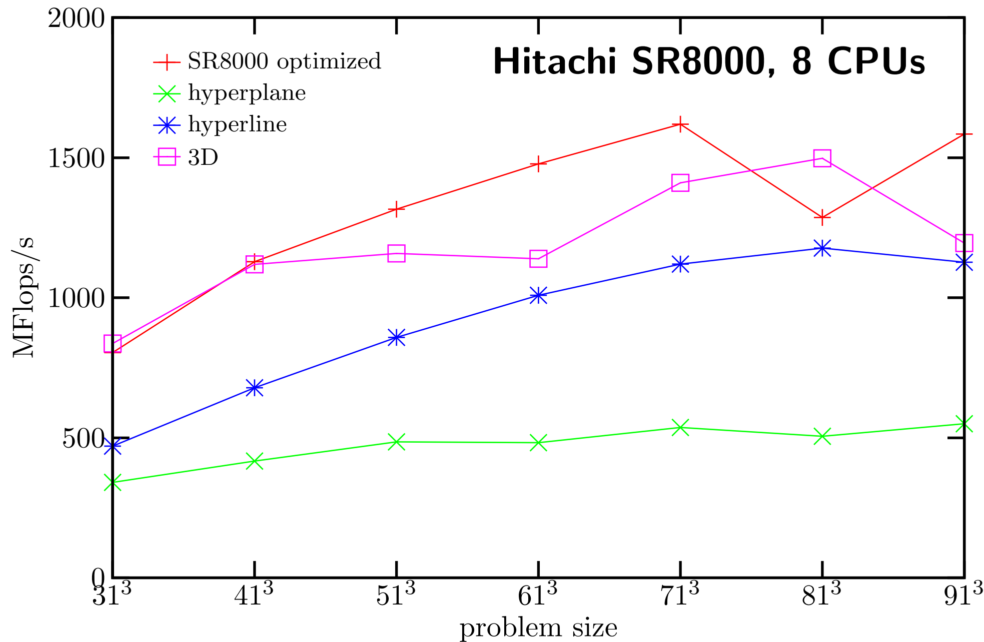

Figure 1.11: Performance for SR8000-optimised version (based on hyperplane-version). For the backward substitution the 3D-indexing was chosen. Besides a `soption nounroll` in front of the forward substitution resulted in a higher level of unrolling and better performance.

However, the Itanium2 workstation shows an increase in performance only up to 2 processors (all 4 CPUs have to share the bandwidth to memory).

1.4.3 Summary

For 1 CPU runs on the SGI Origin 3400, the 3D-version shows performance comparable to the pipeline-parallel-version. For a typical RISC-architecture like the MIPS-CPU this is comprehensible. For higher numbers of CPUs the pipeline-parallel-version performs best. The latter can be observed on every machine except vector-computers.

Intels Itanium2 behaviour is in principle the same as mesured on the SGI. The scaling for the pipeline-parallel-version goes just up to two processors (4 CPUs have to share bandwith).

However, on Power4 the hyperline-version proves to be better than the 3D and the pipeline-parallel version yields the best performance of all for a 1 CPU run. The reason for this is not quite clear yet.

A straightforward port of the hyperplane version to the Hitachi, exploiting the PVP-feature, yields lower performance than the pipeline parallel processing version (3D-version). In spite of the preload mechanism, indirect memory access and varying loop lengths make for slower code. Pipelined parallelism, on the other hand, provides the possibility for outer loop unrolling and cache reuse and is thus the best choice on this machine.

In case of vector architectures, like the Fujitsu VPP300 or the NEC SX5, the hyperplane-version clearly performs best (long inner loops).

In table 1.2 MFlops/s for several machines are given. The appendix contains a description of the benchmark code.

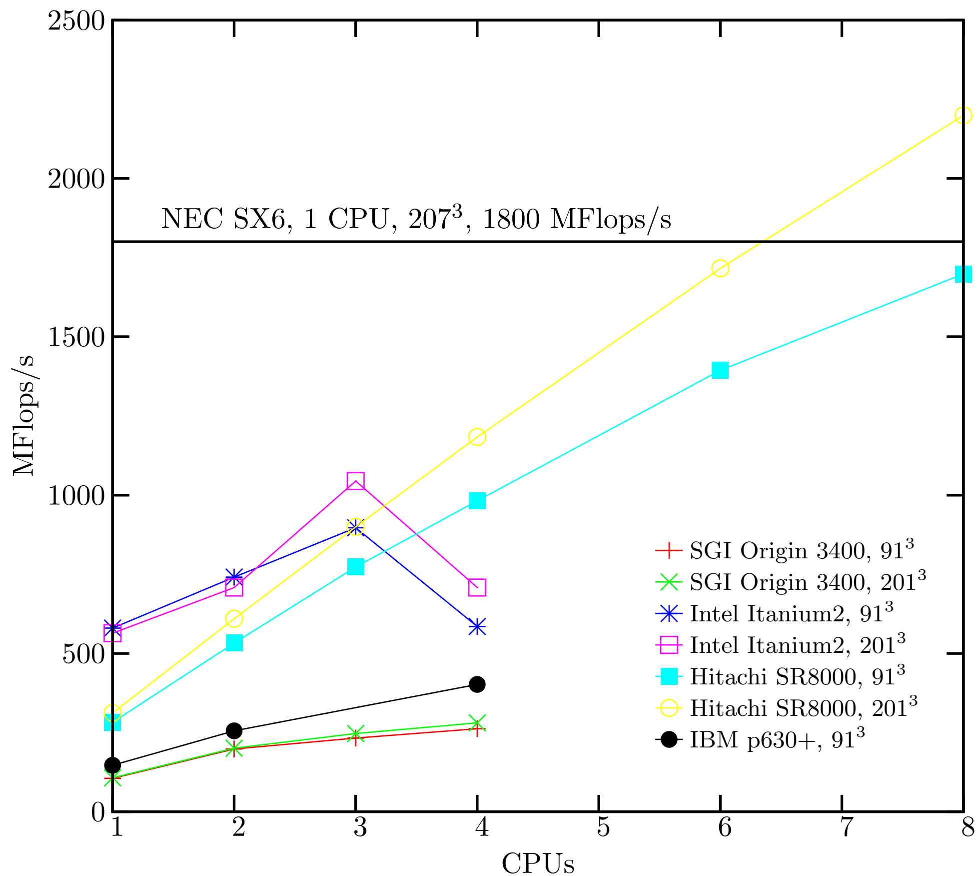

Figure 1.12: Performance for the pipeline-parallel-version (based on 3D-Version). In order to see the effect of system size more clearly a size of 91^3 and 301^3 was choosen.

Table 1.2: Results of SIP-solver benchmark on different architectures. All values are given in MFlops. Calculation of residuals for the 3D-version is integrated in the forward substitution except for the Hitachi SR8000. System size is 91^3 and 500 iterations were performed. A "–" means that measurement were not possible.

	# Proc.	MFlops/s			
		3D	hyperplane	hyperline	ppp
SGI Origin 3400					
	1	117	49	90	106
	2	–	86	129	202
	4	–	130	214	292
	8	–	164	342	416
	16	–	176	530	
Hitachi SR8000-F1					
	1	265	82	185	283
	8	1175	558	1237	1698
opt. Version			1492		
Intel Xeon (2.66 GHz)					
ifc 7.0	1	244	39	197	–
g77 3.2	1	210	38	179	–
Intel Itanium1 (733 MHz)					
	1	203	46	93	–
Intel Itanium2 (1 GHz)					
ifc 7.0 (tiger)	1	554	54	344	578
	2	–	161	513	823
	3	–	260	623	850
	4	–	311	652	709
ifc 7.1 (mc1)	1	592	59	330	523
AMD Opteron (1.6 GHz)					
pgf90 5.0 beta	1	223	84	185	–
gcc 3.3	1	246	67	192	–
ifc (static)	1	262	69	202	–
IBM Power4					
p690, 1.3 GHz	1	162			
p690, 1.3 GHz, opt. Version	1	367			
p655, opt. + large pages	1	633			
p630+, 1.45 GHz	1	137	50	178	147
	2	300	87	318	255
	4	628	148	385	399
NEC SX5e					
	1		745		
NEC SX6					
	1		1800		

A Appendix

A.1 SIP-Solver Benchmark

Installation of the SIP-solver benchmark

The "benchmark package" is called `SipBench.tar.gz`. Unzipping and untaring the archive can be done by invoking

```
gzip -d SipBench.tar.gz
tar -xvf SipBench.tar
```

After that, there is a directory called `SipBench`. Before compiling, the directory has to be entered and an appropriate `make.`*`architecture`*`.inc`-file has to be copied from `sys/` to there. Typing `make` *`size`* `[hyperplane|hyperline|threeD|pwr4|ppp]` creates all specified implementations in the subdirectory `bin/`. There is also a Perl-script available which compiles and runs all different solvers for system sizes of 31^3 to 91^3. For details about compiling and performing measurements, please refer to the README file delivered with the benchmark.

Bibliography

[1] H.L. Stone, *Iterative solution of implicit approximations of multidimensional partial differential equations*, SIAM J. Numerical Analysis, 5 (5), 1968

[2] K. Shimada, T. Kawashimo, M. Hanawa, R. Yamagata and E. Kamada: A Superscalar RISC Processor with 160 FPRs for Large Scale Scientific Processing. Proc. of International Conference on Computer Design (1999), pp. 279–280

[3] J. H. Ferziger, M. Péric, *Computational Methods for FluidDynamics*, Springer Verlag, ISBN 3-540-59434-5, 1999

[4] M. Schäfer, *Numerik im Maschinenbau*, Springer Verlag, ISBN 3-540-65391-0, 1999

List of Figures

1.1 Numbering of nodes for the SIP-Solver. 6
1.2 Hyperplanes in a cube. 10
1.3 Hyperlines in a cube. 13
1.4 Pipeline parallel processing. 18
1.5 Visualisation of a result of a benchmark case (system size 41^3). 20
1.6 SGI Origin 3400: performance of SIP-solver variants. 21
1.7 Hitachi SR 8000: performance of SIP-solver variants. 21
1.8 SGI Origin 3400: performance of SIP-solver versions (single/double precision). 22
1.9 Hitachi: perf. for SIP-solver versions (single/double precision). 23
1.10 SIP-solver benchmark: performance on different architectures. 23
1.11 Performance for SR8000-optimised version. 24
1.12 Performance for the pipeline-parallel-version. 25

List of Tables

1.1 SIP-solver benchmark: problem sizes, number of iterations and floating point operations. 19
1.2 Results of SIP-solver benchmark on different architectures. 26