Rapport de Projet PAP : Jeu de la vie

Albert Truong

Baptiste Sauzet

2 mai 2017

Table des matières

Introduction

Ce projet a été réalisé dans le cadre de l'UE Programmation des Architectures Parallèles de l'Université de Bordeaux. Nous présenterons dans ce rapport les différentes implémentations du sujet qui nous avait été donné : le jeu de la vie de Conway. Ces implémentations diffèrent dans le raisonnement et dans les technologies employés (OpenMP, OpenCl, ...). Nous allons donc dans un premier temps présenter les fonctions que nous utilisons dans la plupart des versions pour appliquer les règles du jeu de la vie, puis nous analyserons le temps d'exécution des fonctions afin d'observer leurs comportements et les comparer en termes d'efficacité.

1 Fonctions annexes

1.1 Détermination de l'état d'une cellule

Nous allons préparer les cellules de l'itération suivante en modifiant `next_img`. Nous appelons pour cela la méthode `will_live` qui prend en paramètre les coordonnées i et j et un booléen `alive` pour indiquer si la cellule est vivante ou non. Nous utilisons cette fonction à plusieurs endroits de notre code, c'est pourquoi nous la détaillons ici.

```
unsigned will_live(unsigned x, unsigned y, bool alive){
  int somme = 0;
  int i, j;
  for(j = -1; j < 2 ; j++)
  {
    for(i = -1; i < 2; i++)
    { // on regarde si la voisine est vivante
      if(cur_img(x+i,y+j) != 0x0 )
        somme += 1;
    }
  }
  if(cur_img(x,y) != 0x0)
    somme -=1;
  if (!alive)
  {
    if(somme == 3) // Resurrection
      return COULEUR;
    return 0x0;
  }
  else
  { // La cellule vit si la somme de ses voisins vaut 2 ou 3
    if(somme == 2 || somme == 3)
      return COULEUR;
    return 0x0;
  }
}
```

Nous allons compter les voisins vivants de la cellule. D'après les règles du jeu de la vie, si une cellule est morte, elle peut revivre si elle a exactement

3 voisins en vie. Si la cellule courante est en vie, elle vivra à la prochaine itération si le nombre de ses voisins en vie vaut 2 ou 3. Ainsi, la fonction retourne la valeur de la cellule à la prochaine itération, `COULEUR` étant la valeur héxadécimale du code couleur de la cellule pour l'affichage de cette dernière.

1.2 Analyse du voisinage

(-1,1)	(-1,0)	(-1,1)
(0,-1)	(0,0)	(0,1)
(1,-1)	(1,0)	(1,1)

FIGURE 1 – Matrice du voisinage

Tout d'abord, nous découpons notre espace en une matrice de dimension 2, comme indiqué sur la figure 1. La cellule courante est à la position *(0,0)*, c'est-à-dire au centre de la matrice. Le reste représente l'ensemble des voisins. Nous vérifions ainsi l'état de chaque cellule, en incrémentant la variable `somme` à chaque fois que l'on croise une cellule vivante dans notre parcours. Pour éviter de mettre des tests à chaque itération pour savoir si nous sommes sur la cellule centrale, nous repassons après le parcours sur la cellule centrale. Si elle est vivante, nous décrémentons `somme` pour ne pas la compter dans le voisinage. Afin de ne pas dépasser du monde de taille `DIM * DIM`, nous partons de la case 1 et nous arrêtons à la case DIM-1 dans le parcours du monde.

2 Versions de Base

2.1 Version séquentielle naïve

Voici le code de notre version séquentielle :

```
unsigned compute_v0 (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it ++)
  {
    for (unsigned i = 1; i < DIM-1; i++)
    {
      for (unsigned j = 1; j < DIM-1; j++)
      {
         if(i != 0 && i != DIM && j != 0 && j != DIM){
           if (cur_img(i,j) == 0) // si la cellule est morte
             next_img(i,j) = will_live(i,j,0);
           else
             next_img(i,j) = will_live(i,j,1);
         }
       }
     }
    swap_images ();
  }
}
```

Les boucles concernant i et j nous permettent de parcourir le monde qui est une matrice de taille `DIM * DIM`. Ensuite, pour chaque cellule, nous allons observer son état, si elle est morte (la case du tableau lui correspondant vaut 0) ou si elle est en vie (la case aura alors la valeur de la couleur attribuée pour l'affichage).

2.2 Version OpenMP for

```
unsigned compute_v1(unsigned nb_iter)
{
  #pragma omp parallel for schedule(static, 16)
  for (unsigned it = 1; it <= nb_iter; it ++)
  ........
```

2.3 Version OpenCL

```
__kernel void test (__global unsigned *in, __global unsigned *out)
{
  int x = get_global_id (0);
  int y = get_global_id (1);
  if (x != 0 && x < DIM-1 && y != 0 && y < DIM-1)
  {
    int somme = 0;
    for (int i = -1; i < 2; i++)
    {
      for (int j = -1; j < 2; j++)
      {
        if (all(color_scatter(in[(y + i) * DIM + (x + j)]) == 0))
        {
          //ne fait rien
        }
        else
        {
          if (i == 0 && j == 0)
          {
          // on ne compte pas la cellule courante
          }
          else
          {
            somme += 1;
            tmp_couleur = in[(y + i) * DIM + (x + j)];
          }
        }
      }
    }
    int result = 0;
    if (all(color_scatter(in[y * DIM + x ]) == 0))
    {
      if (somme == 3)
        result = tmp_couleur;
      else
        result = 0;
```

```
    }
    else
    {
      if (somme == 2 || somme == 3)
        result = tmp_couleur;
      else
        result = 0;
    }
  out[y * DIM + x] = result;
  }
}
```

En ce qui concerne la version OpenCL, nous avons repris le code de la version séquentielle en enlevant les boucles de parcours de l'image qui sont inutiles en OpenCL car chaque pixel est calculé par thread dédié. De plus, nous avons adapté les différents accès mémoire à l'OpenCL.

3 Versions tuilées

3.1 Version séquentielle

```
unsigned compute_v5 (unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it ++)
  {
    for (unsigned i = 1; i < DIM-1; i+=TILEX)
    {
      for (unsigned j = 1; j < DIM-1; j+=TILEY)
      {
        for (unsigned x = i; x < i+TILEX; x++)
        {
          for (unsigned y = j; y < j+TILEY; y++)
          {
            update2(i,j,x,y);
          }
        }
      }
   }
   swap_images ();
```

```
  }
  return 0;
}
```

On parcourt l'image suivant la taille de la tuile puis on parcourt la tuile.

3.2 Version OpenMP for

```
unsigned compute_v6(unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it ++)
  {
    #pragma omp parallel for collapse(2) schedule(static,32)
    for (unsigned i = 1; i < DIM-1; i++)
    {
      for (unsigned j = 1; j < DIM-1; j++)
      {
        if(i != 0 && i != DIM && j != 0 && j != DIM){
          if (cur_img(i,j) == 0) // si la cellule est morte
            next_img(i,j) = will_live(i,j,0);
          else
            next_img(i,j) = will_live(i,j,1);
        }
      }
    } // end parallel for
      swap_images ();
  }
  return 0;
}
```

Dans cette situation, nous faisons effondrer les boucles de i et j via la clause `collapse(2)`, ce qui nous permet de considérer l'ensemble de nos threads comme une matrice en 2 dimensions, que nous identifierons par les coordonnées i et j. Avec la clause `schedule(static,32)`, nous attribuons à chacun de nos threads une tranche de 32 cellules, ce qui correspond à une tuile dans notre cas !, Ainsi, la répartition du travail est égale entre tous les threads.

3.3 Version OpenMp task

```
unsigned compute_v4 (unsigned nb_iter)
```

```
{
  unsigned i, j, x, y;
  for (unsigned it = 1; it <= nb_iter; it ++)
  {
    #pragma omp parallel for collapse(2)
    for ( i = 1; i < DIM-1; i+=TILEX)
    {
      for ( j = 1; j < DIM-1; j+=TILEY)
     {
       #pragma omp task private(x, y)
       {
         for ( x = i; x < i+TILEX; x++)
         ........
```

4 Versions optimisée

4.1 Version séquentielle

```
unsigned compute_v7(unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it ++)
  {
    realIt++;
    for (unsigned i = 1; i < DIM-1; i+=TILEX)
    {
      int indiceI = (i-(1%TILEX)) / TILEX;
      for (unsigned j = 1; j < DIM-1; j+=TILEY)
      {
        int indiceJ = (j-(1%TILEY)) / TILEY;
        if(realIt == 1)
          calculTableauTuile(i,j,indiceI,indiceJ);
        else
        {
          if(tabTuile[indiceI][indiceJ] == 0)
          {
            if(check_neighbors_opti(indiceI, indiceJ) != 0)
              calculTableauTuile(i,j,indiceI,indiceJ);
          }
```

```
            else
              calculTableauTuile(i,j,indiceI,indiceJ);
          }
        }
      }
        swap_images ();
  }
  return 0;
}
```

4.2 Version OpenMP for

```
unsigned compute_v2(unsigned nb_iter)
{
  int indiceI = 0;
  int indiceJ = 0;
  unsigned i, j;
  for (unsigned it = 1; it <= nb_iter; it ++)
  {
    #pragma omp parallel for collapse(2) schedule(static,TILEX) \\
    firstprivate(indiceI,indiceJ) shared(i, j)
    for ( i = 1; i < DIM-1; i+=TILEX)
    {
      for ( j = 1; j < DIM-1; j+=TILEX)
      {
        //printf("avant les indices \n");
        indiceI = (i-(i%TILEX)) / TILEX;
        indiceJ = (j-(j%TILEY)) / TILEY;
        if(realIt == 1)
        ........
```

On a utilisé la version séquentielle optimisée comme base dans laquelle on a ajouté le pragma omp correpondant à la version tuilée. Les nouveaux éléments sont :

— firstprivate : qui permet d'avoir une copie en local.

— shared : qui permet d'avoir en lecture des données qui seront accessibles à de nombreux threads.

4.3 Version OpenMP task

```
unsigned compute_v9(unsigned nb_iter)
{
  for (unsigned it = 1; it <= nb_iter; it ++)
  {
    realIt++;
    #pragma omp parallel for collapse(2)
    for (unsigned i = 1; i < DIM-1; i+=TILEX)
    {
      for (unsigned j = 1; j < DIM-1; j+=TILEY)
      {
        #pragma omp task shared(i,j)
        {
          int indiceI = (i-(1%TILEX)) / TILEX;
          int indiceJ = (j-(1%TILEY)) / TILEY;
          if(realIt == 1)
          ..................
```

En ce qui concerne cette version, nous n'avons pas pu l'ajouter dans les speedups car nous n'avons pas eu assez de temps. Nous avons ajouté le shared(i,j) qui permet à plusieurs threads de lire la même données.

4.4 Version OpenCL

On ce qui concerne cette méthode, nous avons voulu reprendre la méthode employer pour la méthode sequentielle optimisée :

- La création d'un tableau globale de dimension [DIM/tailleTuile][DIM/tailleTuile] qui stockerai le nombre de de cellule vivante.
- La création

Inachevée

5 Versions mixte

Inexistante

6 Comportement observés

Nous n'avons pas eu le temps de générer tous les tests pour observer les comportements, nous nous baserons donc sur nos premières observations et nos hypothèses pour certains cas.

Lorsque l'on regarde tous les speedups,il semblerait que la version OpenCL nous donne un meilleur résultat. La carte graphique gère plus facilement un grand nombre de threads que le CPU, d'où son efficacité. Elle reste performante même lorsqu'il s'agit de calculer la version *4096*4096* du jeu. Cela n'est pas le cas de la version séquentielle par exemple, dont le temps de calcul est exponentiel. La version CPU qui se rapproche le plus de l'efficacité de la version OpenCL est la version openMP `parallel for collapse`, à partir de la version à 1000 itérations.

6.1 Static vs Dynamic

Nous avons exécuté les deux versions mais pas produit tous les graphes. Nous avons cependant remarqué que la version static est plus performante au fur et à mesure que la matrice croît. Nous pensons que cela vient d'un problème de localité mémoire. C'est-à-dire qu'en dynamic, les threads vont récupérer un travail à effectuer dès qu'ils ont fini le travail qu'ils avaient en cours. On perd ainsi l'ordre d'exécution des calculs, qui peuvent par chance arriver au sein d'une zone mémoire quasi-contiguë, plus précisément dans la même ligne de cache, et ainsi éviter des accès mémoires trop coûteux. A l'inverse, les calculs peuvent aussi amener à multiplier les défauts de cache, en effectuant les tâches dans des zones mémoires qui ne sont pas préchargées dans le cache.

On peut en déduire que la version *static* préserve l'exécution du programme de trop de défauts de caches, risques auxquels s'expose la version *dynamic*

6.2 Coût du parallélisme

Lors de l'exécution des versions de taille 256, nous remarquons que certaines versions OpenMP prennent plus de temps que la version séquentielle (la version openMP de base en *static* par exemple). Cela se remarque notamment lorsqu'il y a peu d'itérations (cf courbes en annexe). Le parallélisme est déjà rentable en *dynamic* mais il faut attendre d'avoir une *DIM* de taille

*1024*1024* pour observer la rentabilité du *static*, toujours dans la version openMP de base.

Cependant, la version séquentielle voit sa complexité augmenter de manière exponentielle par rapport à l'augmentation de la taille de la matrice, au profit des versions parallélisées qui vont utiliser plus de ressources du processeur (contre un coeur physique seulement pour la version séquentielle), et donc freiner l'augmentation du temps d'exécution du programme.

Annexes

Voici dans l'ordre les PDF de comparaison de temps d'exécution de nos versions : Nos versions sont numérotées ainsi :

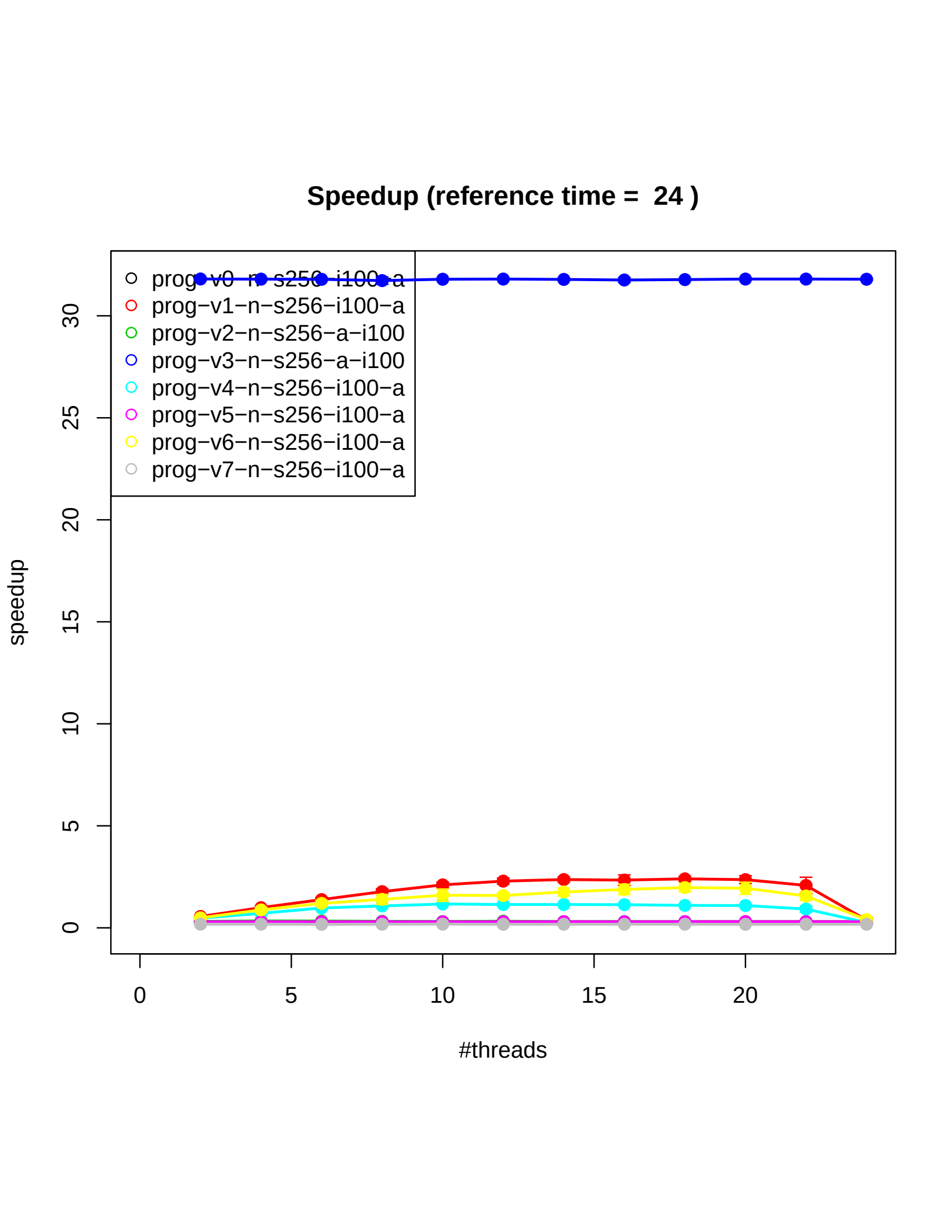
- v0 : Séquentielle Naïve
- v1 : OpenMP Naïve par défaut schedule(dynamic,16), sinon spécifié static 32
- v2 : OpenMP Zone (optimisée)
- v3 : OpenCL
- v4 : OpenMP Task
- v5 : Séquentielle tuilée
- v6 : OpenMP tuilée (collapse)
- v7 : Séquentielle optimisée

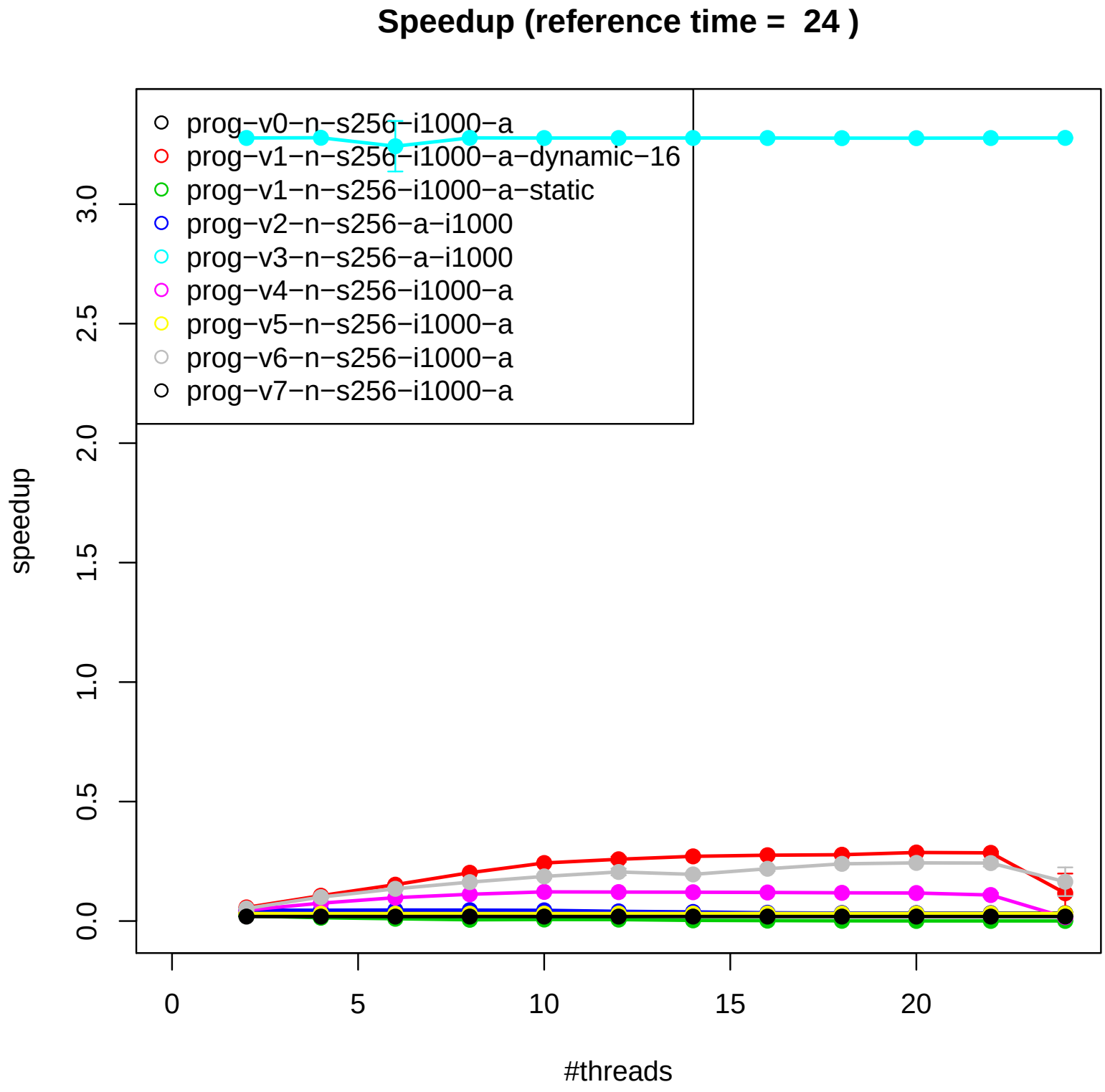
Nous n'avons pas ajouté la version open MP task optimisée dans les speedups par manque de temps.

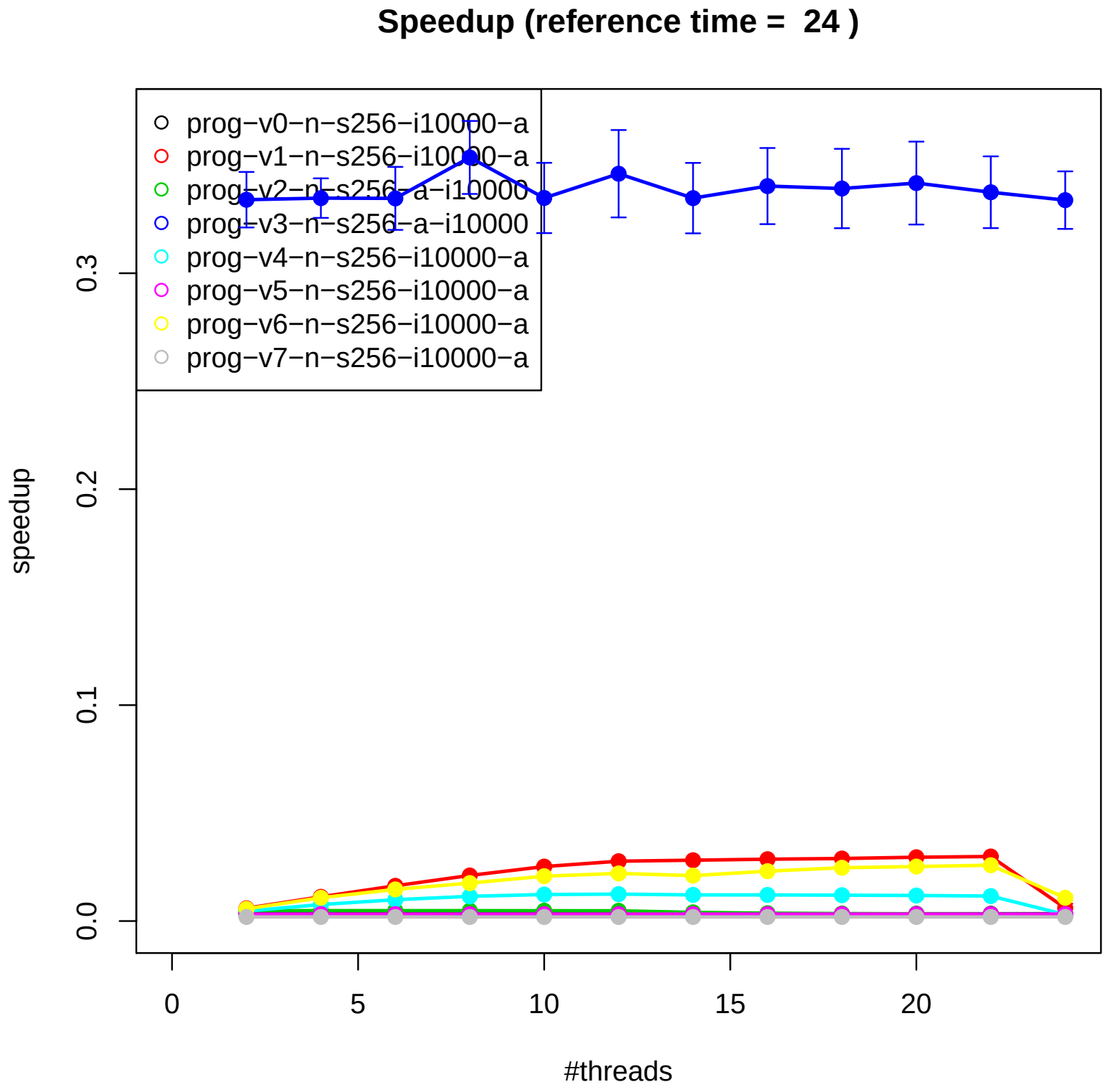
Dans l'ordre nous retrouverons :

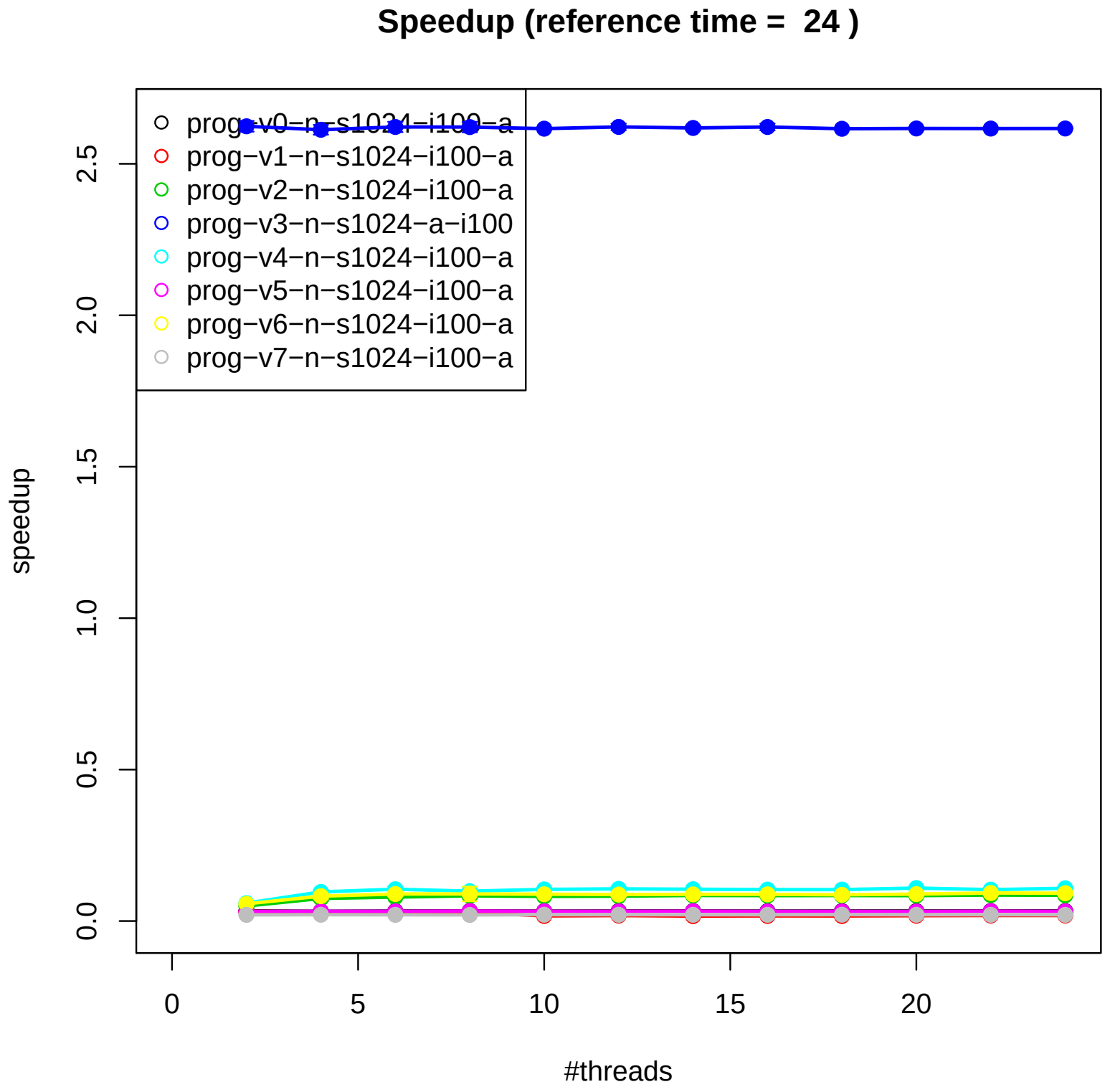
- Taille tuile : 32 -s 256 -i 100
- Taille tuile : 32 -s 256 -i 1000
- Taille tuile : 32 -s 256 -i 10000
- Taille tuile : 32 -s 1024 -i 100
- Taille tuile : 32 -s 1024 -i 1000
- Taille tuile : 32 -s 1024 -i 10000
- Taille tuile : 32 -s 4096 -i 1000
- Taille tuile : 32 -s 4096 -i 10000
- **Taille tuile :16** :
- Taille tuile : 16 -s 256 -i 100 avec opencl
- Taille tuile : 16 -s 256 -i 100 sans opencl
- Taille tuile : 16 -s 256 -i 1000 sans opencl

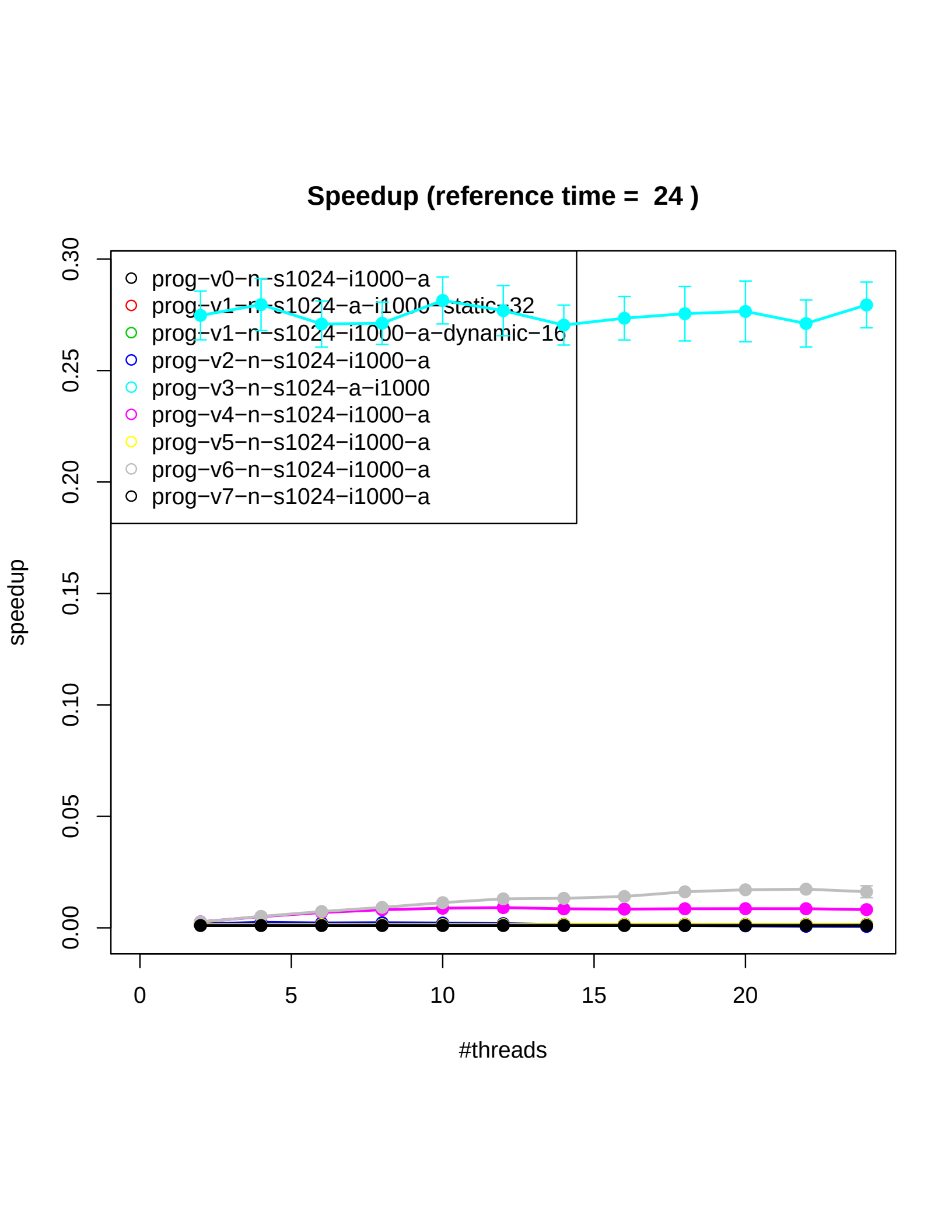
— Taille tuile : 16 -s 256 -i 10000 sans opencl

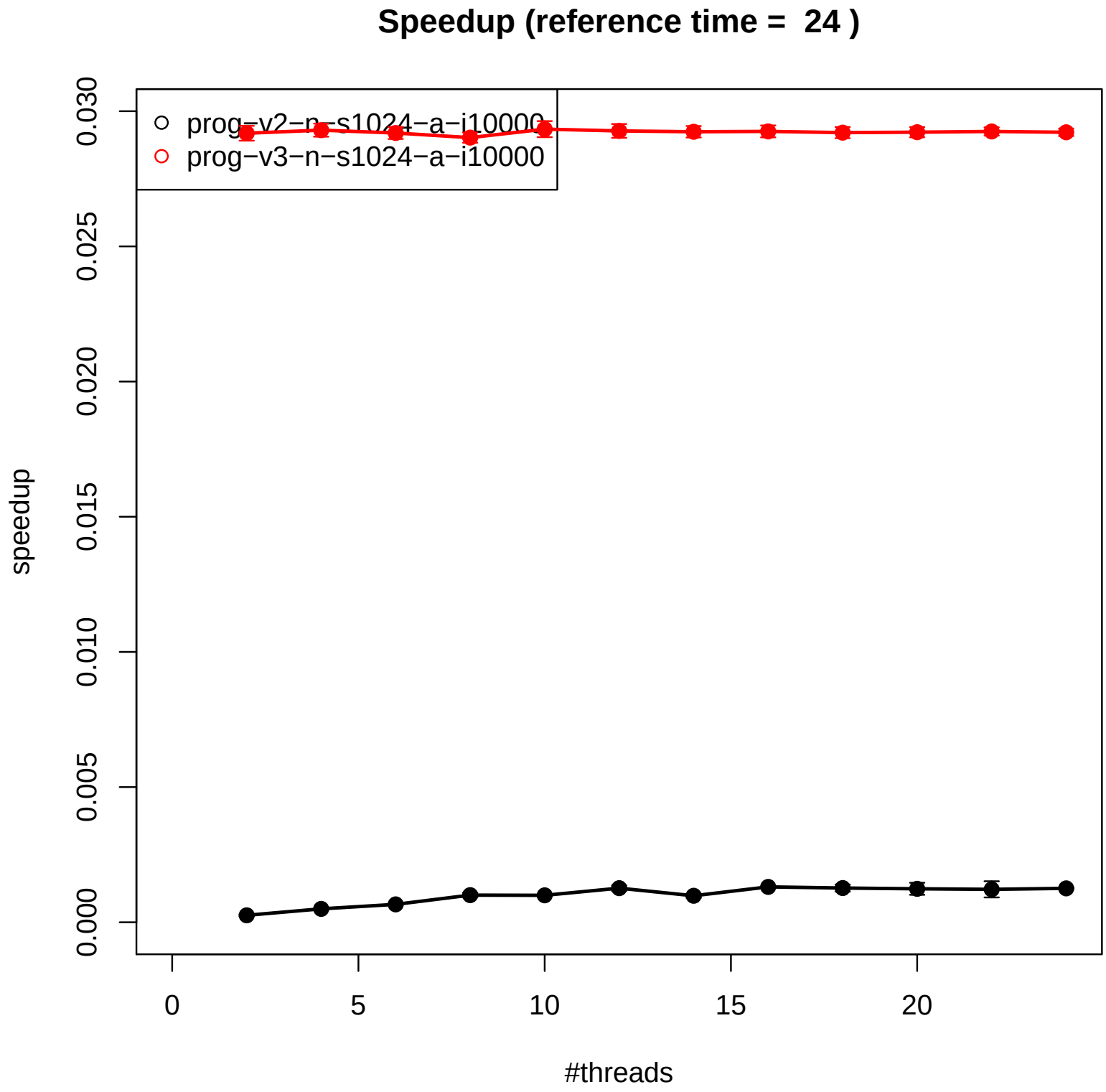

Speedup (reference time = 24)
prog−v0−n−s256−i100−a
prog−v1−n−s256−i100−a
prog−v2−n−s256−a−i100
prog−v3−n−s256−a−i100
prog−v4−n−s256−i100−a
prog−v5−n−s256−i100−a
prog−v6−n−s256−i100−a
prog−v7−n−s256−i100−a
speedup
0
5
10
15
20
25
30
0
5
10
15
20
#threads

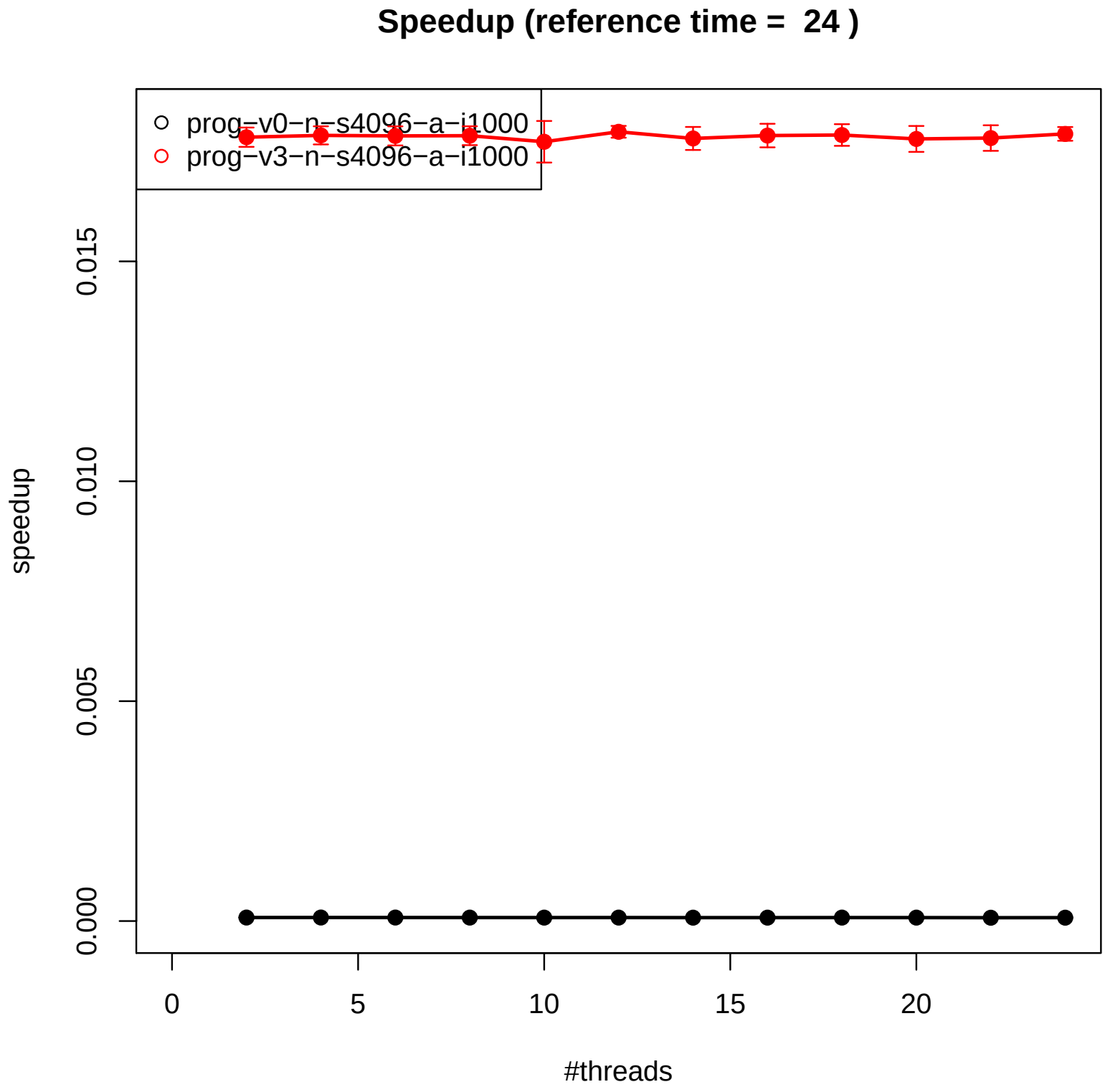

Speedup (reference time = 24)
prog−v0−n−s256−i1000−a
prog−v1−n−s256−i1000−a−dynamic−16
prog−v1−n−s256−i1000−a−static
prog−v2−n−s256−a−i1000
prog−v3−n−s256−a−i1000
prog−v4−n−s256−i1000−a
prog−v5−n−s256−i1000−a
prog−v6−n−s256−i1000−a
prog−v7−n−s256−i1000−a
speedup
0.0
0.5
1.0
1.5
2.0
2.5
3.0
0
5
10
15
20
#threads

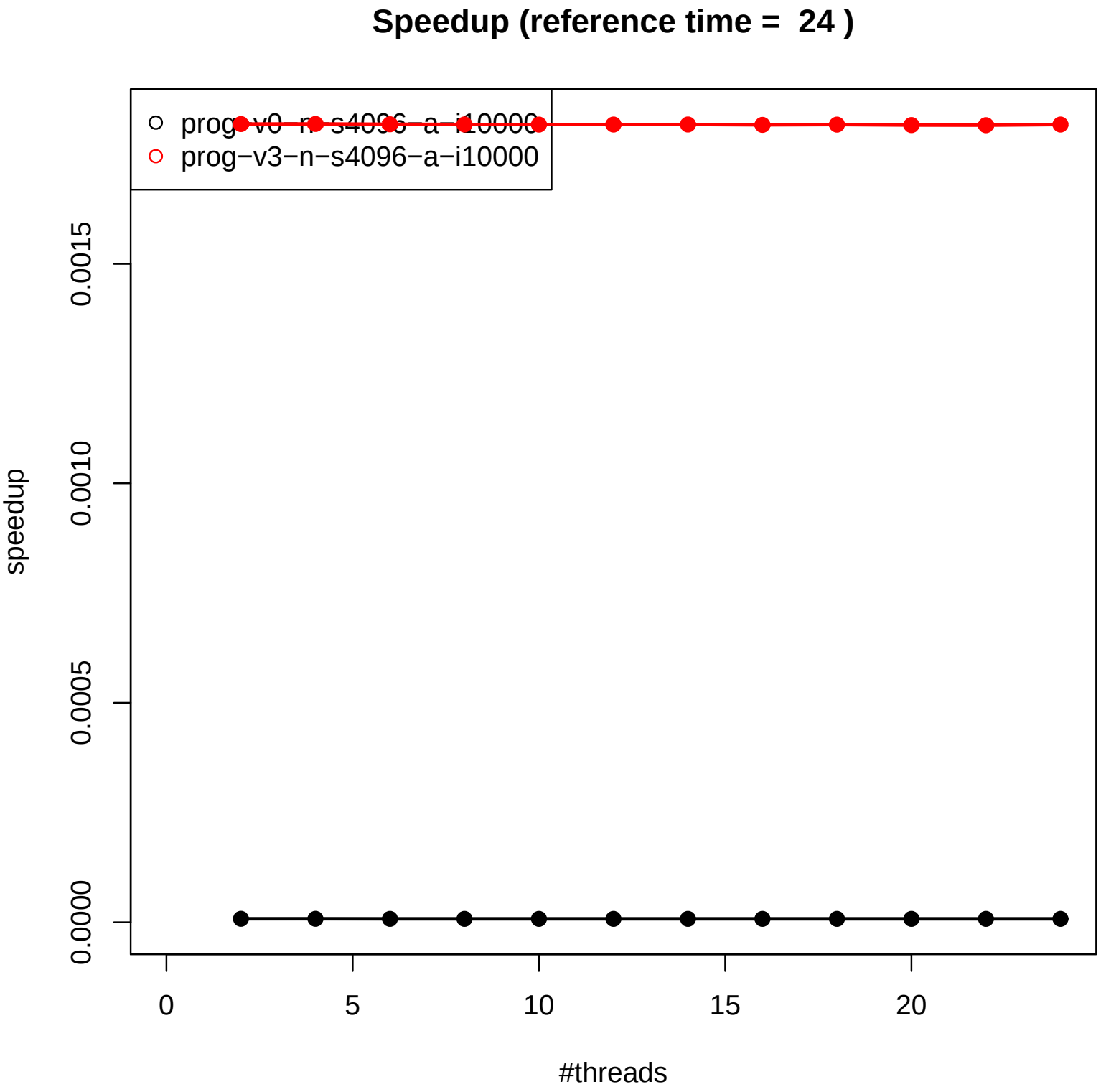

Speedup (reference time = 24)
prog−v0−n−s256−i10000−a
prog−v1−n−s256−i10000−a
prog−v2−n−s256−a−i10000
prog−v3−n−s256−a−i10000
prog−v4−n−s256−i10000−a
prog−v5−n−s256−i10000−a
prog−v6−n−s256−i10000−a
prog−v7−n−s256−i10000−a
speedup
0.0
0.1
0.2
0.3
0
5
10
15
20
#threads

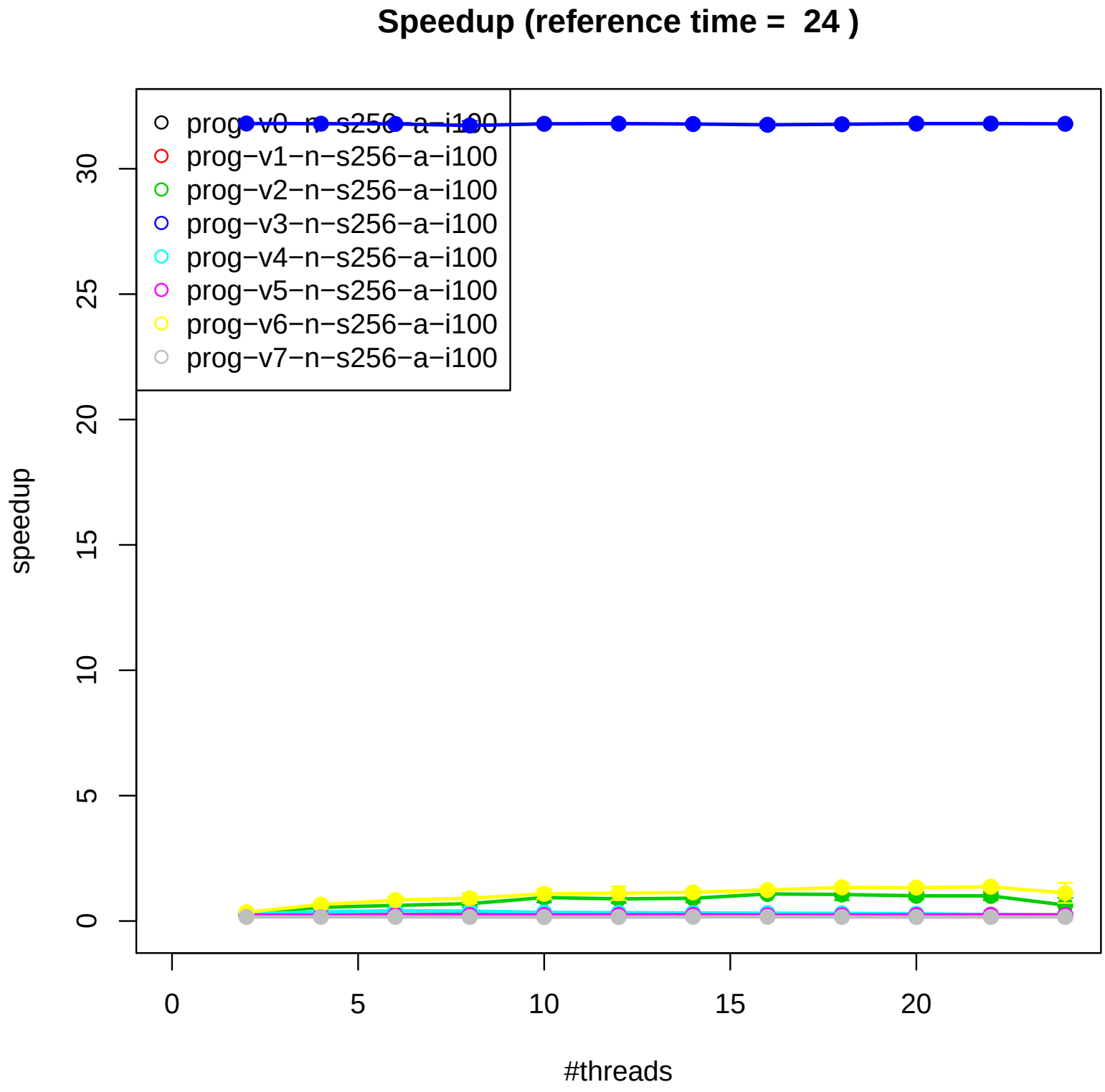

Speedup (reference time = 24)
prog−v0−n−s1024−i100−a
prog−v1−n−s1024−i100−a
prog−v2−n−s1024−i100−a
prog−v3−n−s1024−a−i100
prog−v4−n−s1024−i100−a
prog−v5−n−s1024−i100−a
prog−v6−n−s1024−i100−a
prog−v7−n−s1024−i100−a
speedup
0.0
0.5
1.0
1.5
2.0
2.5
0
5
10
15
20
#threads

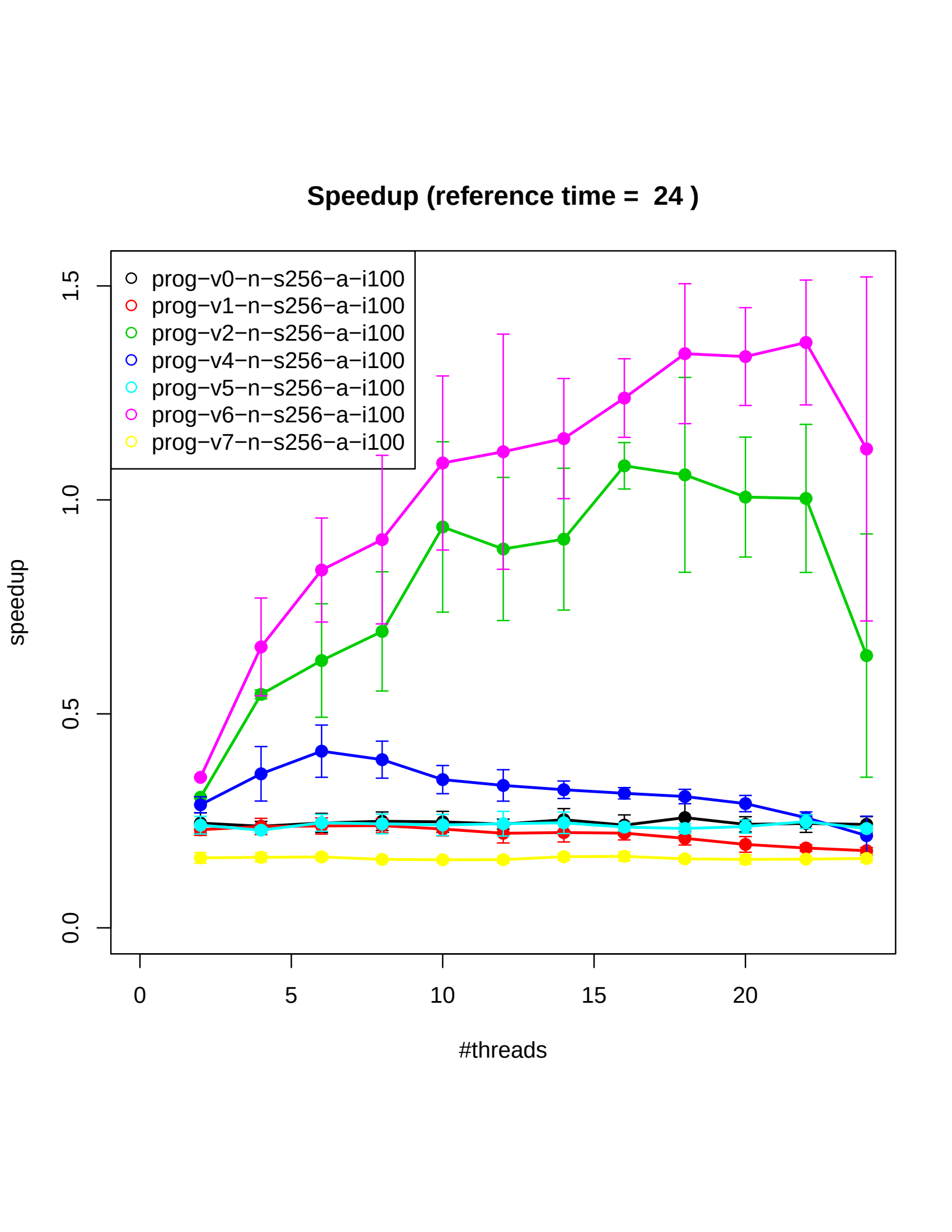

Speedup (reference time = 24)
prog−v0−n−s1024−i1000−a
prog−v1−n−s1024−a−i1000−static−32
prog−v1−n−s1024−i1000−a−dynamic−16
prog−v2−n−s1024−i1000−a
prog−v3−n−s1024−a−i1000
prog−v4−n−s1024−i1000−a
prog−v5−n−s1024−i1000−a
prog−v6−n−s1024−i1000−a
prog−v7−n−s1024−i1000−a
speedup
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0
5
10
15
20
#threads

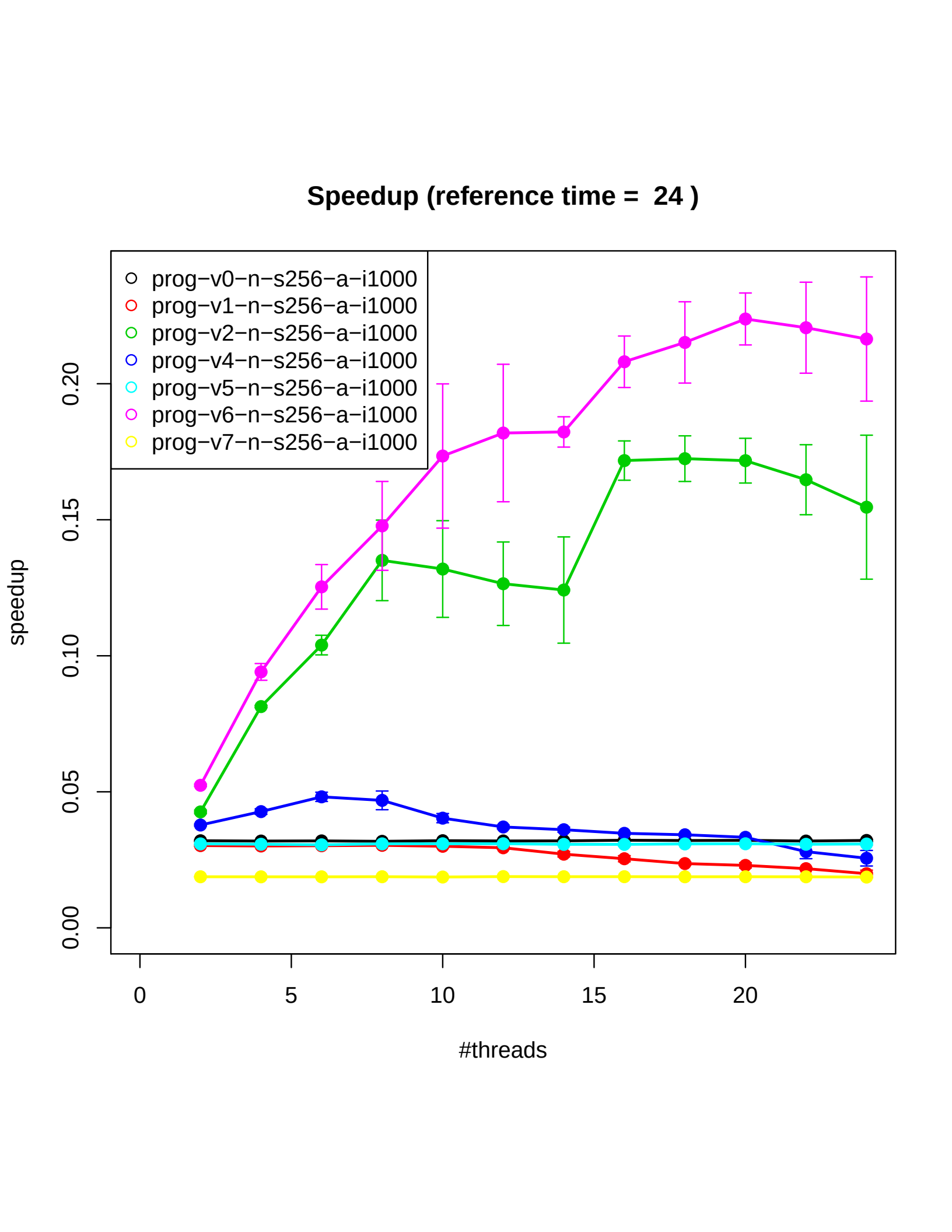

Speedup (reference time = 24)
prog−v2−n−s1024−a−i10000
prog−v3−n−s1024−a−i10000
speedup
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0
5
10
15
20
#threads

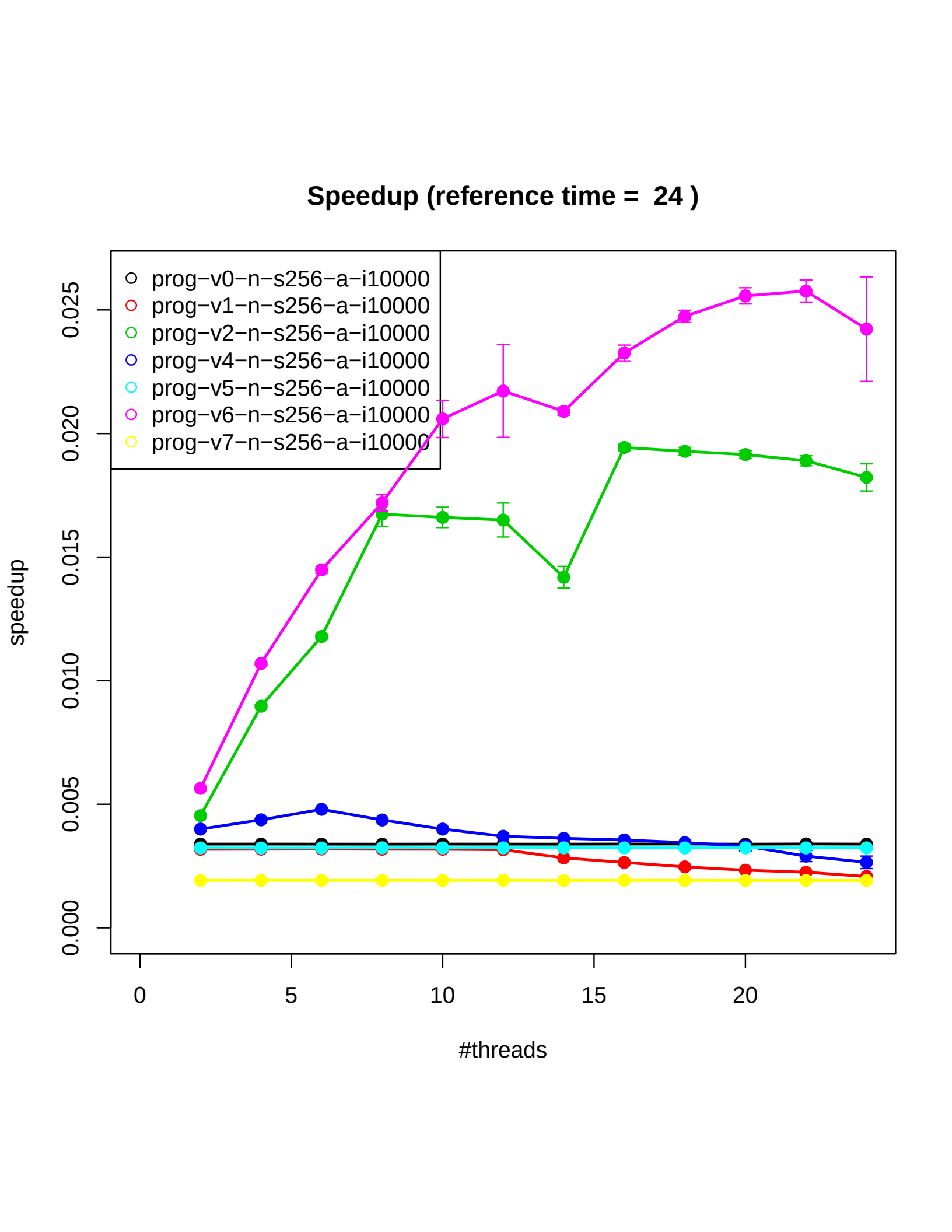

Speedup (reference time = 24)
prog−v0−n−s4096−a−i1000
prog−v3−n−s4096−a−i1000
speedup
0.000
0.005
0.010
0.015
0
5
10
15
20
#threads


Speedup (reference time = 24)
prog−v0−n−s4096−a−i10000
prog−v3−n−s4096−a−i10000
speedup
0.0000
0.0005
0.0010
0.0015
0
5
10
15
20
#threads


Speedup (reference time = 24)
prog−v0−n−s256−a−i100
prog−v1−n−s256−a−i100
prog−v2−n−s256−a−i100
prog−v3−n−s256−a−i100
prog−v4−n−s256−a−i100
prog−v5−n−s256−a−i100
prog−v6−n−s256−a−i100
prog−v7−n−s256−a−i100
speedup
0
5
10
15
20
25
30
0
5
10
15
20
#threads


Speedup (reference time = 24)
prog−v0−n−s256−a−i100
prog−v1−n−s256−a−i100
prog−v2−n−s256−a−i100
prog−v4−n−s256−a−i100
prog−v5−n−s256−a−i100
prog−v6−n−s256−a−i100
prog−v7−n−s256−a−i100
speedup
0.0
0.5
1.0
1.5
0
5
10
15
20
#threads


Speedup (reference time = 24)
prog−v0−n−s256−a−i1000
prog−v1−n−s256−a−i1000
prog−v2−n−s256−a−i1000
prog−v4−n−s256−a−i1000
prog−v5−n−s256−a−i1000
prog−v6−n−s256−a−i1000
prog−v7−n−s256−a−i1000
speedup
0.00
0.05
0.10
0.15
0.20
0
5
10
15
20
#threads


Speedup (reference time = 24)
prog−v0−n−s256−a−i10000
prog−v1−n−s256−a−i10000
prog−v2−n−s256−a−i10000
prog−v4−n−s256−a−i10000
prog−v5−n−s256−a−i10000
prog−v6−n−s256−a−i10000
prog−v7−n−s256−a−i10000
speedup
0.000
0.005
0.010
0.015
0.020
0.025
0
5
10
15
20
#threads